Filipe André Luís Ribeiro de Carvalho

Licenciado em Engenharia Informática

Tratamento pela periferia de falhas na Internet

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : José Augusto Legatheaux Martins, Prof. Catedrático,
Universidade Nova de Lisboa

Júri:

Presidente:	Doutor Nuno Manuel Robalo Correia
Arguente:	Doutor Ricardo Lopes Pereira
Vogal:	Doutor José Augusto Legatheaux Martins

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2012

Tratamento pela periferia de falhas na Internet

À família e amigos.

Agradecimentos

Os meus mais sinceros agradecimentos ao Professor José Legatheaux Martins, por todo o apoio e atenção prestado no desenrolar deste trabalho, sem o qual este não poderia estar tão completo e estruturado. Os meus agradecimentos à Faculdade de Ciências e Tecnologia (FCT) e principalmente ao departamento de Informática por todo o apoio logístico e de infra-estruturas que cedeu, que sem esse apoio os resultados experimentais teriam sido mais complicados de obter. Agradecer ao departamento de Informática, mais propriamente ao Professor João Moura Pires pela bolsa de investigação SAIA (Sistema de Análise de Informação Académica) que me permitiu continuar e completar os estudos no presente ano lectivo.

O meu obrigado a todos os colegas que me ajudaram durante este período, em especial ao Bruno Filipe Faustino por ter ajudado a encontrar um nome para o mecanismo, à minha colega Tânia Ferreira Leitão por ter ajudado em revisões sempre que precisei e ao colega João Saramago por me ter inserido na tecnologia *Java Native Interface* (JNI) (utilizada neste trabalho).

Um obrigado em especial à minha amiga Tânia Ferreira Leitão por estar sempre disponível e disposta a ajudar, pelas boleias e paciência mostrada, e aos meus amigos Bruno Filipe Faustino, Sérgio Silva e André Fidalgo por estarem sempre presentes quando precisei. Um obrigado ao meu amigo José Carlos Simões Figueiredo por estar sempre disposto a me ouvir.

Um obrigado à minha irmã, aos meus pais e avós, que sem eles o meu percurso teria sido muito mais difícil.

Resumo

O crescimento da Internet acabou por evidenciar um conjunto de limitações que o desenho da mesma possui, tanto ao nível do encaminhamento como da arquitectura. O protocolo de encaminhamento usado (i.e. *Border Gateway Protocol*) apresenta deficiências para lidar com os actuais requisitos da Internet, que se agravam cada vez mais, levando a que o tempo de convergência do sistema em caso de falhas se esteja a tornar elevado e difícil de tratar. A juntar aos problemas inerentes ao protocolo, existem cada vez mais sistemas que possuem mais que um caminho para um determinado destino (e.g. *multi-homing* ou várias interfaces). Esta multiplicidade de caminhos, agrava o número de entradas nas tabelas de encaminhamento, pondo em risco o bom funcionamento do encaminhamento na Internet, mas ao mesmo tempo incrementa a diversidade e as alternativas de encaminhamento. Apesar de existirem propostas que tentam solucionar os problemas existentes, nos mais variados níveis arquitecturais, estas acabam por exigir mudanças demasiado complexas e incomportáveis.

Este trabalho tem por objectivo encontrar formas de desenvolver aplicações distribuídas, num contexto em que os computadores intervenientes têm acesso a múltiplos endereços IP origem e/ou destino, de forma a usar essa diversidade de caminhos (que é em si positiva) para mascarar eventuais falhas de encaminhamento da Internet e, adicionalmente, tentar explorar o melhor dos caminhos possíveis. Trata-se, portanto, de uma solução do nível aplicação que pressupõe apenas que é possível usar diferentes pares de endereços para comunicar com cada interlocutor.

Esta solução permitirá criar aplicações que explorem os diversos caminhos disponíveis, contornando os problemas do protocolo actual de encaminhamento e, se possível, melhorando as velocidades de transferência das conexões.

Palavras-chave: Internet, BGP, Diversidade, Nível Aplicação

Abstract

Internet growth has brought to light a set of limitations, both at an architectural level and at a forwarding level. The protocol used for routing packets on the Internet (Border Gateway Protocol) has problems dealing with the requirements of today's Internet, which are getting worse, making the time that the system needs to stabilize in case of link failures elevated and difficult to deal with. Adding to these problems, more and more systems are getting access to multiple paths to a certain destination (e.g. multi-homing or more than one interfaces). These multiple paths increases the number of entries in the routing tables, putting at risk the entire packet forwarding system in the Internet. At the same time these multiple paths increases diversity and forwarding alternatives. Despite the existence of several proposals, in all the architectural levels, that try to solve the existing problems, they end up being to complex to be implemented worldwide.

The objective of this work is to find ways to develop distributed applications, in a context which participating computers have access to multiple origin and/or destination IP addresses, in a way that they can use this diversity of paths to bypass eventual forwarding failures and, if possible, using the best path available. The ultimate goal is to use path diversity to our advantage, allowing us to compensate convergence failures and the system capacity to deal with them. This is a solution at the application level which proposes that it is possible to use different pairs of addresses to communicate between each end of the communication.

This solution will allow us to develop applications that can explore multiple paths, bypassing forwarding problems and, if possible, increasing connection's speed.

Keywords: Internet, BGP, Diversity, Application Level

Conteúdo

1 Introdução **1**

1.1 Objectivos 3

1.2 Estrutura do resto do documento 3

2 Estado da arte do encaminhamento entre domínios **5**

2.1 *Transmission Control Protocol* 5

2.2 *Border Gateway Protocol* 6

2.2.1 Funcionamento do BGP 8

2.3 O BGP perante a escala da Internet 9

2.3.1 Aumento no número de mensagens de actualização 11

2.3.2 Crescimento das tabelas de encaminhamento 12

2.3.3 Balanceamento de Carga 13

2.4 Melhoramentos a curto prazo 15

2.5 Sumário 15

3 Outros modelos de encaminhamento e escolha de caminhos **17**

3.1 Novos esquemas de encaminhamento 18

3.1.1 Feedback Based Routing 18

3.1.2 NIRA 19

3.2 *Locator Identifier Separation Protocol* 21

3.2.1 Mapeamento de EIDs em RLOCs 23

3.2.2 Funcionamento 24

3.2.3 Vantagens e Desvantagens 25

3.2.4 LISP Mobile Node 27

3.3 *MultiPath TCP* 28

3.4 Sumário 29

4 Proposta **31**

4.1 Objectivos da proposta 33

4.2 Especificação da proposta 34
4.2.1 Nome, intenção e motivação 35
4.2.2 Casos de aplicabilidade do padrão desenvolvido 35
4.3 Discussão preliminar das hipóteses subjacentes à proposta 37
4.3.1 O cliente tem diversos endereços 37
4.3.2 O servidor tem diversos endereços 37
4.3.3 Interacção entre o cliente e o servidor baseada em TCP com várias conexões 38
4.4 Sumário 38

5 Concretização do padrão em Java **41**
5.1 Ferramentas auxiliares 45
5.1.1 *WEB10G* 45
5.1.2 *Java Native Interface* 47
5.2 Estrutura 48
5.3 Principais funcionalidades 52
5.3.1 Inicialização do *NChannelSocket* 52
5.3.2 Controle de histórico de conexões 57
5.3.3 Escolha inicial de um par de endereços 57
5.3.4 Escolha posterior de outro par de endereços 58
5.3.5 Probing 58
5.3.6 Decisão de mudança 59
5.4 Implementação do padrão 63
5.5 Implementação concreta do padrão 66
5.6 Sumário 67

6 Avaliação e Resultados **69**
6.1 Ambientes de teste e configuração 69
6.2 Cenários de teste 73
6.2.1 Controlo 73
6.2.2 Teste 1 - Comutação sem histórico 79
6.2.3 Teste 2 - Comutação com histórico 81
6.2.4 Teste 3 - O melhor canal fica indisponível 82
6.2.5 Teste 4 - O melhor canal fica com pior *performance* 86
6.2.6 Teste 5 - Teste de duas interfaces na origem 91
6.3 Sumário 95

7 Conclusões e trabalho futuro **97**
7.1 Conclusões 97
7.2 Trabalho futuro 99

Lista de Figuras

2.1 iBGP no mesmo AS e eBGP entre ASes [Cis] 7

3.1 Exemplo de um pacote LISP em formato IPv4 [brk04] 22
3.2 Exemplo do funcionamento de LISP em Unicast [brk04] 24

5.1 Diagrama de pacotes 49
5.2 Diagrama de classes 50
5.3 Diagrama de arquitectura e execução 51

6.1 Configuração da rede no ambiente de teste 1 71
6.2 Modelo do ambiente de teste 1 72
6.3 Configuração da rede no ambiente de teste 2 72
6.4 Teste 3 - Exemplo de transferência 87
6.5 Teste 4 - Exemplo de transferência 90

Lista de Tabelas

2.1 Perfil do BGP entre os dias 4 e 10 de Dezembro de 2011 [bgp11]. 11
2.2 Dados no AS 65000 entre Outubro 2011 e Dezembro 2011. 12

4.1 Atributos principais de um padrão segundo [GHJV95] 34

5.1 Interacção padrão Socket Java 41
5.2 Interacção padrão pedido/resposta 43
5.3 Interacção padrão pedido/resposta continuado com teste de qualidade . . 43
5.4 Interacção padrão incluindo tratamento de excepções 44
5.5 Variáveis web10g aplicadas no padrão 46

6.1 Características dos canais usando *Dummynet* 70
6.2 Tamanho do bloco de 32768 bytes 74
6.3 Tamanho do bloco de 131072 bytes 74
6.4 Tamanho do bloco de 524288 bytes 75
6.5 Tamanho do bloco de 1048576 bytes 75
6.6 Tamanho do bloco de 20 MB - Totalidade do ficheiro um só pedido 76
6.7 Número de pedidos ao servidor por tamanho do bloco 76
6.8 Controlo usando pior canal 77
6.9 Controlo usando canal intermédio 78
6.10 Controlo individual de cada canal do modelo 1 - usando *NChannelSocket* . 78
6.11 Overhead do mecanismo de sondas 79
6.12 Teste 1 - Comutação sem histórico 80
6.13 Teste 2 - Comutação com histórico 81
6.14 Teste 3 - Melhor canal fica indisponível, *timeout* 500ms 83
6.15 Teste 3 - Melhor canal fica indisponível, *timeout* 1s 84
6.16 Teste 3 - Melhor canal fica indisponível, *timeout* 5s 85
6.17 Novas características da conexão melhor 86
6.18 Teste 4 - Melhor canal piora, tamanho do bloco de 512KB 88

6.19 Teste 4 - Melhor canal piora, tamanho do bloco de 128KB 89
6.20 Teste 5 - Controlo 91
6.21 Teste 5 - canal Ethernet fica indisponível, *timeout* 500ms 92
6.22 Teste 5 - canal Ethernet fica indisponível, *timeout* 1s 93
6.23 Teste 5 - canal Ethernet fica indisponível, *timeout* 5s 94

Listagens

5.1 Interacção padrão Socket Java 42
5.2 Interacção padrão Socket Java 43
5.3 Interacção padrão pedido/resposta continuado com teste de qualidade . . 43
5.4 Interacção padrão incluindo tratamento de excepções 44
5.5 Exemplo de um método nativo 47
5.6 Construtor 1 53
5.7 Construtor 2 54
5.8 Construtor 3 55
5.9 Construtor 4 56
5.10 Decisão de mudança 61
5.11 Implementação do Java Socket 64
5.12 Implementação do Java NChannelSocket 64

Lista de Acrónimos

ACK *Acknowledgment*. 5, 6

ALT *Alteranative-Topology*. 23–27

AS *Autonomous System*. 1, 2, 7–10, 12, 13, 15, 18, 19

BGP *Border Gateway Protocol*. 2, 3, 5–15, 17–19, 21, 23, 25, 26

CDN *Content Delivery Network*. 32, 38

DNS *Domain Name System*. 14, 20, 24, 31, 32, 38, 52, 53

eBGP *External BGP*. 7, 9, 15

EID *EndPoint Identifier*. 21–27, 29

ETR *Egress Tunel Routers*. 22–24, 26, 27

FIB *Forwarding Information Base*. 8, 12

HTML *HyperText Markup Language*. 6, 36

HTTP *Hypertext Transfer Protocol*. 31, 38, 63, 73

iBGP *Internal BGP*. 7, 15

IP *Internet Protocol*. 1–3, 5–9, 14, 18–26, 31, 32, 46, 53, 70, 71, 82

ISP *Internet Service Provider*. 1, 2, 7, 10, 14, 19, 32, 37

ITR *Ingress Tunnel Routers*. 22–27

JNI *Java Native Interface*. vii, 45, 47, 67

LISP *Locator Identifier Separation Protocol*. 3, 17, 21–29, 32, 37

LISP-MN *LISP Mobile Node*. 27, 28

MPTCP *Multipath TCP*. 3, 17, 28, 29, 33

MSS *Maximum Segment Size*. 6, 45, 46, 60

MTU *Maximum Transmission Unit*. 26

NAT *Network Address Translation*. 14, 21, 25, 28, 32, 37

P2P *Peer to Peer*. 32, 33

QoS *Quality of Service*. 2, 10

RIB *Routing Information Base*. 8, 12, 15

RIR *Regional Internet Register*. 1

RLOC *Routing Locator*. 21–27, 29

RTO *Retransmission timer*. 6, 45, 46, 50, 57, 79

RTT *Round-trip delay time*. 6, 18, 45, 46, 50, 59, 70, 71, 86

TCP *Transmission Control Protocol*. 5–8, 10, 11, 17, 18, 27–29, 33, 36–39, 41, 44–46, 74, 97

TIPP *Topology Information Propagation Protocol*. 19, 20

UDP *User Datagram Protocol*. 22, 23

1

Introdução

Nos meados de 1960 o exército Norte-Americano criou uma rede (ARPANET), que tinha o propósito de servir de sistema de comunicação entre as várias organizações governamentais dos Estados Unidos da América. Esta rede cedo despertou interesse por parte de diversas organizações e companhias, que pretendiam um sistema de partilha de informação semelhante.

Com o passar dos anos, cada vez mais organizações e universidades aderiram a esta iniciativa, criando grupos de redes ligadas entre si que permitem a partilha de informação variada através da troca de pacotes de dados, encaminhados por um algoritmo especifico ou seja, a Internet. O crescimento da Internet foi exponencial até 2001 (e superlinear desde então) [DD08], o que acabou por evidenciar um conjunto de limitações que o desenho da mesma possui, tanto ao nível do encaminhamento como ao nível da arquitectura.

A Internet actual é composta por um conjunto cada vez maior de sistemas autónomos - *Autonomous Systems* (ASes). Um AS pode ser visto como uma organização que gere as suas redes privadas e que está ligado a outros ASes, providenciando aos seus clientes (caso existam) o acesso à Internet a nível global. Os ASes que providenciam serviços, nomeadamente de trânsito de pacotes, são controlados por *Internet Service Providers* (ISPs)[1], sendo que cada ISP pode controlar mais que um AS.

A cada AS é atribuído (e.g. pelo *Regional Internet Register* (RIR)[2] onde se encontra) um número que o identifica univocamente perante o encaminhamento e um ou mais prefixos *Internet Protocol* (IP)[3] específicos. Verificam-se dois tipos de ASes, os de trânsito e os de

[1]Organização que controla um ou mais ASes e que providência serviços a clientes, consoante um certo número de políticas de negócio, tanto entre os clientes como entre outros ISPs.

[2]Entidade responsável pela gestão de nomes e números numa determinada zona do globo.

[3]Um endereço IP é representado por 32 bits (IPv4) ou 128 bits (IPv6), separado por secções de igual valor.

acesso. Um AS de trânsito é aquele que permite que tráfego originado ou destinado a outros ASes flua através de si. Os ASes de acesso apenas permitem que tráfego direccionado a si flua na sua rede. O núcleo da Internet é assegurado por ASes de trânsito que se unem, garantindo a espinha dorsal do encaminhamento a nível mundial.

A comunicação (i.e. a troca de mensagens) entre ASes é feita utilizando um protocolo de encaminhamento, denominado de *Border Gateway Protocol* (BGP) (Capítulo 2). Este protocolo tem sofrido algumas alterações ao longo dos anos, de forma a se adaptar às condições, cada vez mais complexas, da Internet. Estas modificações têm-se revelado suficientes a curto prazo, mas não resolvem os problemas de fundo do protocolo como, a falta de controlo de tráfego ou a inexistência de qualidade de serviços - *Quality of Service* (QoS).

Quando este protocolo foi pensado, desenhado e implementado, existiam poucos ASes e a Internet (como a conhecemos agora) estava ainda a dar os seus primeiros passos. Contudo, verificou-se o aumento no número de ASes, não só pelo desenvolvimento natural da Internet, mas principalmente devido à quantidade de organizações que se conectaram a mais que um provedor de serviços (ISP), com o intuito de aumentar a fiabilidade das suas conexões e a distribuição do seu tráfego. Estes ASes são denominados de *multi-homed* e têm que obter os seus próprios prefixos IP, assim como anunciar um prefixo por cada ligação que possuam a um ISP, de modo a poderem participar nas actividades de encaminhamento entre os domínios dos diversos ISPs a que estão ligados. Esta adesão em massa das organizações a múltiplos ISPs veio aumentar, não só o número de ASes que existe, mas também o número de prefixos IP existente nas tabelas de encaminhamento dos *routers*.

Este aumento do número de entradas nas tabelas de encaminhamento, aliado a problemas inerentes ao próprio protocolo faz com que, em caso de falhas, o tempo que demora a rede a voltar a um estado coerente (i.e. o tempo de convergência do sistema) seja cada vez maior, encontrando-se em alguns casos nas dezenas de minutos. Em última análise, os problemas com o BGP estão a afectar directamente o encaminhamento no núcleo da Internet. Por outro lado, o protocolo não permite explorar adequadamente a diversidade de caminhos existentes na Internet, não suporta engenharia de tráfego nem *multi-homing* de forma adequada.

Os problemas encontrados no encaminhamento actual abrem portas à criação de novos protocolos de encaminhamento, ou mesmo a arquitecturas de Internet totalmente novas. Existe uma linha de pensamento em particular que visa dar à periferia (aos sistemas fora do núcleo) mais liberdade na escolha do encaminhamento, actuando como uma escapatória a parte dos problemas presentes na Internet. Pretende-se tirar partido do número, cada vez maior, de ASes *multi-homed*, de modo a estes aproveitarem as diversas ligações existentes sem agravar ainda mais o estado do encaminhamento. Esta liberdade

Por exemplo, um endereço IPv4 é representado por A.B.C.D /n, onde /n é chamada a dimensão do prefixo IP. Esta dimensão do prefixo identifica o número de bits significativos utilizados para identificar uma rede. Por exemplo, 192.9.205.22 /18 significa que os primeiros 18 bits identificam a rede e os restantes 14 bits os computadores.

pode ser aproveitada tanto a nível de novas arquitecturas (NIRA [YCB07] e *Feedback Based Routing* [ZGC03]), como em propostas ao nível transporte (*Multipath TCP* (MPTCP) [RHF09]) ou mesmo ao nível rede (*Locator Identifier Separation Protocol* (LISP) [brk04]).

Algumas das propostas anteriores materializam-se, do ponto de vista dos computadores da periferia, no facto de estas terem acesso a vários endereços IP distintos. Tal é também uma realidade sempre que os computadores têm várias interfaces distintas, reais ou virtuais (i.e baseadas em túneis).

O acesso a diferentes endereços origem/destino para que os computadores comuniquem cria oportunidades suplementares para explorar a diversidade do encaminhamento e até para mascarar as eventuais falhas, ou outras deficiências menos graves do BGP.

1.1 Objectivos

Este trabalho tem por objectivo encontrar formas expeditas de desenvolver aplicações distribuídas, num contexto em que os computadores intervenientes têm acesso a múltiplos endereços IP origem e/ou destino, de forma a usar essa diversidade de caminhos para mascarar eventuais falhas de encaminhamento da Internet e, complementarmente, tentar explorar o melhor dos caminhos possíveis.

Trata-se, portanto, de uma solução do nível aplicação que propõe apenas que é possível usar diferentes pares de endereços para comunicar com cada interlocutor.

Para facilitar o desenvolvimento das aplicações, a solução é disponibilizada na forma de um padrão de programação em uma linguagem de alto nível (*Java* [jave]), complementado por um *framework* que permite guiar o processo de escolha do melhor caminho, i.e. escolher para qual comutar e fazer a comutação, e acelerar a velocidade de selecção do mesmo.

1.2 Estrutura do resto do documento

Em seguida apresenta-se a estrutura do resto deste documento.

No Capítulo 2 é feita uma apresentação do estado do encaminhamento actual na Internet (i.e. o seu protocolo, o BGP), a sua configuração, os seus problemas e algumas soluções a curto prazo que tentam minorar esses mesmos problemas. No Capítulo 3 expomos várias alternativas ao encaminhamento actual: o *Feedback Based Routing*; o NIRA; o LISP; e o MPTCP, todas elas tendo em comum dar maior poder à periferia no que toca às decisões de encaminhamento. No Capítulo 4 descrevemos em detalhe o que foi feito, i.e. a especificação dos objectivos propostos, assim como desenvolvemos alguns casos de aplicabilidade do padrão desenvolvido e de várias hipóteses subjacentes à proposta. No Capítulo 5 explicamos detalhadamente como concretizámos o padrão, explicando as ferramentas auxiliares utilizadas. Neste capítulo encontra-se definida toda a estrutura

de classes criada assim como as principais funcionalidades implementadas. Apresentamos também exemplos de uma implementação concreta utilizando Java Socket, e uma implementação concreta do nosso padrão. No Capítulo 6 fazemos a avaliação do padrão e *framework* desenvolvido, fazendo no total 5 conjuntos de testes, separados por dois ambientes distintos. Finalmente, no Capítulo 7, apresentamos as conclusões deste trabalho assim como todas as questões relacionadas com o trabalho futuro.

2

Estado da arte do encaminhamento entre domínios

Neste capítulo iremos explicar como funciona actualmente o encaminhamento entre os diferentes domínios na Internet e os problemas que este levanta.

O resto deste capítulo está estruturado da seguinte forma: na próxima secção (Secção 2.1) fazemos uma breve introdução a um dos protocolos de transporte utilizados na Internet, o *Transmission Control Protocol* (TCP), pela importância que apresenta para a compreensão deste trabalho; na Secção 2.2 é explicado o funcionamento do protocolo de encaminhamento entre domínios utilizado actualmente, o *Border Gateway Protocol* (BGP); na Secção 2.3 são discutidos como os principais problemas presentes na Internet hoje em dia estão a afectar o normal funcionamento do protocolo e, por consequência, da própria Internet; por fim, na Secção 2.4, são explicadas algumas soluções a curto prazo que pretendem minorar esses problemas.

2.1 *Transmission Control Protocol*

O protocolo TCP [SC91] é o principal protocolo de transporte utilizado na Internet. Na secção seguinte e na restante parte do trabalho estaremos bastante dependentes das suas características, pelo que esta secção resume as suas principais características.

O protocolo tem como uma das suas principais características a capacidade de detectar perca de pacotes IP [Posa] e garantir que pacotes fora de ordem cheguem ao destinatário pela ordem correcta.

Para ajudar na detecção de pacotes perdidos é utilizada uma técnica que consiste no receptor responder com mensagens de *Acknowledgment* (ACK) aos pacotes recebidos. A

parte que envia mantém um registo de cada pacote enviado e aguarda pelo respectivo ACK para o considerar recebido. Associado a cada pacote enviado existe também um temporizador denominado *Retransmission timer* (RTO) [AP] que, findo esse tempo sem obter resposta (ACK), o pacote é dado como perdido e é retransmitido. Este mecanismo de envio e recepção de pacotes depende do *Round-trip delay time* (RTT). O RTT [AP] é o tempo que leva um certo pacote a ser enviado, somado ao tempo que leva a receber uma resposta. Este valor está envolvido no cálculo de várias variáveis associadas a uma conexão, como é o caso do *Smoothed RTT* (média ponderada utilizada no cálculo do RTO) e o próprio RTO.

Enquanto que o IP trata do encaminhamento dos pacotes, o TCP mantém um registo das diversas unidades transmitidas, chamadas de segmentos, nas quais uma mensagem foi dividida para um encaminhamento eficiente pela rede. Por exemplo: quando um ficheiro *HyperText Markup Language* (HTML) [htm] é pedido de um servidor, o TCP divide o ficheiro em segmentos e encaminha-os individualmente para a camada IP. Esta encapsula cada segmento TCP num pacote IP adicionando um cabeçalho que, entre outras, inclui o endereço IP destino. Ao receber os pacotes, a camada TCP monta os segmentos individuais, assegurando que estão por ordem e sem erros, enquanto os envia para a aplicação.

Este protocolo tem também como grande trunfo o facto de conseguir lidar com casos de congestionamento na rede. Implementa 4 algoritmos importantes [Ste] e que estão interligados. Aquando do estabelecer de uma conexão TCP, os dois pontos da conexão anunciam o tamanho das respectivas janelas, i.e. o número máximo de bytes que uma das das partes está preparada para receber. Acordam também um valor para o tamanho máximo que um segmento terá na rede, denominado de *Maximum Segment Size* (MSS) [Posb]. Através do algoritmo *Slow Start*, uma nova janela (janela de congestionamento) é criada com o tamanho de um segmento. Sempre que um ACK é recebido o tamanho desta janela é incrementada em um segmento. Eventualmente este crescimento da janela chegará a um ponto em que estão a ser enviados mais dados do que a capacidade da rede, sendo que os *routers* intermédios começam a descartar pacotes. É nesta situação que se diz que o tamanho da janela de congestionamento atingiu uma dimensão insuportável para a rede.

O protocolo incorpora diversos algoritmos para computar dinamicamente o valor da janela de congestionamento em função da história da troca de pacotes anterior. Por esta razão a janela de congestionamento é um indicador importante do estado da conexão, dando-nos uma indicação de que se a conexão está num estado inicial ou de recuperação (janela inferior ou igual a $2 * MSS$) [SAP] ou num estado mais estável.

2.2 *Border Gateway Protocol*

O BGP é o protocolo que suporta as decisões de encaminhamento no núcleo da Internet. Baseia-se em um algoritmo distribuído que, aplicado no contexto da Internet, utiliza

prefixos de endereços IP como unidade básica de encaminhamento.

Desde a sua implementação, o BGP passou por várias versões. A mais notável, e que é ainda hoje a utilizada, é a versão 4 de 1996, onde foi introduzido o encaminhamento sem classes, denominado CIDR[1].

O BGP utiliza algoritmos de vector-distância para disseminar informação de alcance (*reachability*) entre ASes. Como qualquer protocolo de encaminhamento vector-distância, o BGP opera com um sincronismo muito fraco baseado em informação parcial [HA06]. Na realidade, o BGP seria melhor classificado como sendo um protocolo vector-caminhos, já que, como iremos ver em seguida, a noção de distância é substituída pela de caminho completo em termos dos ASes a atravessar.

Existem dois tipos de BGP [Phi06] (Figura 2.1), o *Internal BGP* (iBGP) e o *External BGP* (eBGP).

eBGP

Utilizado para troca de informação entre ASes vizinhos, através de sessões TCP [SC91] entre os diversos *routers*. As mensagens trocadas são: de criação das ligações TCP e BGP; de actualização de caminhos; de *keep-alive* periódicos para vigiar as ligações; de notificação, em caso de ser necessário reportar erros ou fechar as ligações BGP.

iBGP

Utilizado por *routers* BGP (iBGP), pertencentes ao mesmo AS, para partilha de informação. O iBGP é muito utilizado no caso de grandes ISPs, onde estes possuem mais que uma saída da sua rede, utilizando o iBGP para orientar o tráfego com destino externo, dentro do seu AS.

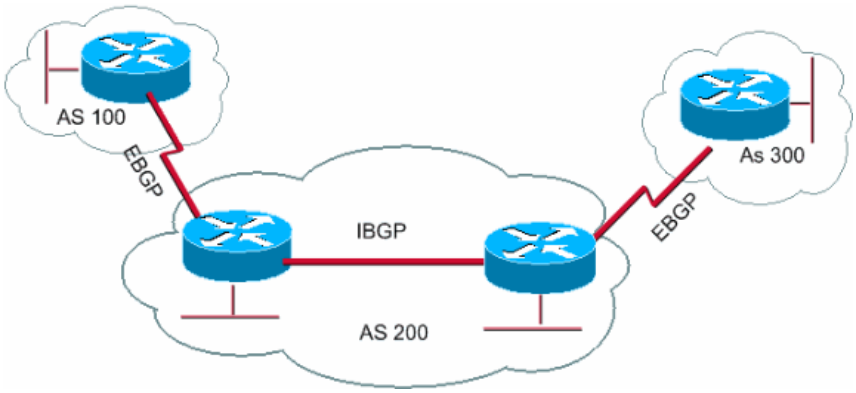

Figura 2.1: iBGP no mesmo AS e eBGP entre ASes [Cis]

Daqui em diante, sempre que surgir a sigla BGP, não se especificando a que tipo de BGP se refere, admite-se que se fala sempre de eBGP. Os detalhes do funcionamento do iBGP não se enquadram no domínio deste estudo.

[1] *Classless Inter-Domain Routing* - Sistema de encaminhamento onde a dimensão dos prefixos IP deixaram de estar fixos por classes, para passarem a ser de tamanho livre.

2.2.1 Funcionamento do BGP

Internamente, cada *router* BGP mantém duas tabelas, a *Forwarding Information Base* (FIB) e a *Routing Information Base* (RIB). A RIB contem todos os caminhos conhecidos pelos *routers* vizinhos, enquanto que a FIB contem apenas o melhor caminho utilizado pelo *router* até cada destino. Este caminho é computado com o auxilio do algoritmo de selecção de rotas sobre a tabela RIB. Para além destas duas tabelas, o *router* BGP continua a possuir a de encaminhamento propriamente dita.

Para comunicarem, os *routers* BGP estabelecem uma ligação TCP entre eles enviando, a partir daí, mensagens de actualização. Estas mensagens de actualização podem ser de dois tipos: mensagens de anúncio (*Advertisement*), com os destinos (prefixos) e caminhos entre ASes que, segundo o algoritmo de selecção de rotas, são os melhores; mensagens de supressão (*Withdrawal*), enviadas em caso de falhas.

Quando um *router* BGP recebe uma mensagem de actualização, se esta não for invalidada por filtros internos à política do AS, o BGP executa um conjunto de acções dependendo do tipo de mensagem que recebeu:

- se for uma mensagem de anúncio, a rota recebida substitui a que está presente na FIB se for melhor que a que lá se encontra. Caso contrário, a rota apenas é adicionada à RIB;
- se for uma mensagem de supressão, a rota é retirada da RIB e, se estiver presente na FIB, é também retirada desta. Ao ser removida da FIB, o algoritmo de selecção de rotas escolhe a próxima melhor das disponíveis na RIB ou nenhuma, caso não possua alternativas.

Após processada uma mensagem de actualização, uma ou mais mensagens são propagadas para os vizinhos se:

- o algoritmo de selecção de rotas considerar que a rota que recebeu é melhor, em relação ao que estava na tabela de encaminhamento, para o mesmo prefixo IP;
- não há rota para o destino, enviando mensagens de supressão.

Como o BGP não suporta encaminhamento multi-caminho, se mais que um caminho for considerado como óptimo, é aplicado um algoritmo de selecção de rotas, para seleccionar qual deve ser utilizado e inserido na FIB.

O funcionamento do BGP está directamente relacionado com um sistema de encaminhamento baseado em politicas (*policy routing*) entre os diversos ASes, sendo que isso também influencia as escolhas do que considera como sendo o melhor caminho. Os atributos mais importantes para escolher a melhor rota são listados em seguida.

Preferência local

Atributo local a um AS que indica uma preferência por determinado caminho em relação a outro. Pode ser visto como uma espécie de *ranking* de caminhos.

Caminho entre ASes (AS *Path*)

O caminho entre ASes contem todos os números de todos os ASes que constituem o caminho entre a origem da mensagem e o destino (i.e. um caminho é um vector com o número de ASes num dado caminho, em que casa posição tem o número que identifica o AS). É utilizado para evitar ciclos e pode também ser utilizado para aplicar políticas de encaminhamento entre ASes.

Próximo salto (*Next hop*)

IP do próximo *router* no caminho entre ASes.

***Multi Exit Discriminator* (MED)**

Actua como uma espécie de sugestão entre dois ASes de modo a um poder indicar que caminho prefere para receber uma mensagem, quando mais que um caminho óptimo existe.

Origem

Descreve como um *router* aprendeu um determinado caminho. Pode ser IGP (se aprendeu dentro do próprio AS), EGP (se foi aprendido através do eBGP) ou mesmo *Incomplete* (se foi aprendido de outro modo ou de forma desconhecida).

Comunidades

Um caminho pode conter uma ou mais comunidades que podem ser vistas como grupos de características comuns que pretendem efectuar as mesmas regras de transmissão e disseminação das mensagens de encaminhamento do BGP.

O processo de selecção segue os seguintes passos:

1. Seleccionar a rota com valor de preferência local mais alto;
2. Seleccionar a rota com o caminho entre ASes mais curto;
3. Seleccionar a rota com o atributo MED mais baixo, se as rotas foram recebidas do mesmo AS;
4. Seleccionar a rota com a métrica IGP mais baixa, e.g. com menor custo ou menor distância entre *routers*, dependendo da implementação do IGP que o AS possui;
5. Aplicar outra regra, por exemplo: escolher o IP mais baixo entre os disponíveis para o próximo salto.

2.3 O BGP perante a escala da Internet

O BGP está longe de ser perfeito, possuindo muitos problemas inerentes ao próprio protocolo, como:

Escalabilidade

Não é simples garantir que um algoritmo distribuído, como é o caso do BGP, seja escalável numa rede com a dimensão da Internet. A quantidade, cada vez maior, de prefixos existentes nas tabelas de encaminhamento e o aumento do número de

mensagens de actualização geradas actualmente é algo que dificulta a escalabilidade deste algoritmo.

Convergência

O BGP, em caso de falha de acesso a um AS, é um algoritmo que rapidamente inunda a rede com mensagens de supressão o que, por sua vez, aumenta significativamente o tempo de convergência da rede, ou seja, o tempo que demora até a rede estabilizar novamente. Por outro lado, quando uma rota é suprimida, as alternativas conhecidas pelos *routers* podem não ser as melhores, pelo que pode ter que vir a corrigir o melhor caminho, voltando a propagar actualizações até que tudo estabilize.

Qualidade das rotas

Ao basear o seu funcionamento em um algoritmo de vector-distância e as distâncias virem expressas em número de ASes atravessados, o BGP consegue simplicidade no funcionamento à custa de rotas com pouca qualidade. Nada garante que uma rota onde um pacote atravesse apenas dois sistemas autónomos seja melhor (e.g. mais rápida) do que outra em que atravesse mais que dois ASes.

Balanceamento de carga

Balanceamento de carga, ou de tráfego, tanto o que entra como aquele que sai de um determinado *router*, é a capacidade de escolher e/ou dividir por que canal irá passar o tráfego, quando existem múltiplos caminhos. A falta de suporte nativo a este problema está cada vez a fazer-se sentir mais no bom funcionamento do BGP, principalmente com o rápido aumento do número de redes *multi-homed*. Este tópico é discutido em detalhe na Subsecção 2.3.3.

Qualidade de serviço (QoS)

O BGP não possui suporte a qualidade de serviço. QoS implica, por exemplo: definir prioridades de acesso ou garantir certos bit-rates ou rácio de perca de pacotes num determinado sistema/AS/ISP. A falta de consenso neste aspecto entre os diversos sistemas autónomos também não contribuiu para o seu desenvolvimento.

Segurança

Não existe qualquer nível de segurança no BGP que impossibilite a um determinado sistema autónomo anunciar, através do BGP, prefixos que não sejam os dele, propositadamente ou não. Não existe também qualquer mecanismo de prevenção contra ataques à ligação TCP. Como a ligação BGP mantém-se enquanto se mantiver a ligação TCP, este ataque poderia levar a um frenesim de actualizações.

Se considerarmos estes problemas, mas agora na escala global e com a complexidade que a Internet actual possui, cedo se percebe que estamos a convergir para uma situação delicada. O tamanho, cada vez maior, das tabelas de encaminhamento e do número de mensagens de actualização a circular entre *routers* BGP está a afectar todo o sistema, sendo estes os dois problemas que se podem revelar mais críticos a curto prazo. Problemas como o da segurança serão minorados com o uso de S-BGP (*Secure-BGP* [Ken03]).

Tabela 2.1: Perfil do BGP entre os dias 4 e 10 de Dezembro de 2011 [bgp11].

Descrição	Valor
Número de mensagens BGP de actualização	888.121
Número de actualizações de prefixos	1.990.743
Número de supressões de prefixos	108.995
Número médio de prefixos por mensagem de actualização	2,36
Número médio de mensagens de actualização por segundo	1,28
Número médio de prefixos actualizados por segundo	3,24
Pico do número de mensagens de actualização por segundo	1.200 (14:17:59 de dia 7)
Pico de prefixos actualizados por segundo	410 (20:57:13 de dia 7)
Pico de prefixos suprimidos por segundo	5.625 (15:51:08 de dia 7)
Número de prefixos	392.797
Número de ASes de origem	39.653
Número de caminhos únicos	197.596

2.3.1 Aumento no número de mensagens de actualização

Uma sessão BGP usa conexões TCP para transporte de dados. Como as comunicações são fiáveis, não é necessário recalcular periodicamente toda a tabela de encaminhamento e, consequentemente, todos os canais de comunicação, como acontece com outros protocolos (e.g. RIP2 [Mal]). Assim sendo, o BGP torna-se um protocolo menos dispendioso em termos de recursos já que, após a comunicação TCP/BGP estar estabelecida, apenas se trocam mensagens de actualização [HA06].

Se a Internet fosse totalmente estável, então as únicas mensagens trocadas seriam as de *keep-alive*, de 30 em 30 segundos (por omissão), simplesmente a indicar que estava tudo operacional. Contudo, a Internet não é estável e, com a criação e a perda de caminhos, rapidamente se gera um grande número de mensagens de actualização num curto espaço de tempo.

O BGP provoca assim grandes instabilidades na rede em caso de falhas. Se um *router* BGP estiver mal configurado ou mal gerido poderá entrar num ciclo de mudança de estados. Este ciclo provoca um envio constante de mensagens de supressão e de anúncio de rotas, conhecido por *route flapping* [CRR98]. O *route flapping* provoca geralmente uma actividade excessiva em todos os *routers* já que, os caminhos estão constantemente a serem anunciados e removidos das tabelas de encaminhamento. Aliado a isso, quanto mais tempo demorar um *router* a processar as mensagens de actualização, mais tempo leva todo o sistema a convergir e mais mensagens se acumulam nos *routers*, aumentando a carga de processamento e podendo levar o tempo de convergência de todo o sistema a vários minutos.

No caso extremo, um *router* acabaria por esgotar o *buffer* de mensagens BGP, falhando o processamento das mensagens. Neste caso, instalar-se-ia a inconsistência na informação partilhada, podendo levar ao colapso do sistema de encaminhamento [HA06].

Tabela 2.2: Dados no AS 65000 entre Outubro 2011 e Dezembro 2011.

Dados presentes na FIB / RIB	Outubro 2011	Novembro 2011	Dezembro 2011
Entradas BGP activas (FIB)	377.870	382.683	386.857
Todas as entradas BGP (RIB)	746.514	756.260	765.140
Rácio RIB/FIB (746514/377870)	1,9756	1,9762	1,9778

Através dos valores apresentados na Tabela 2.1 é perceptível a complexidade e o volume de tráfego que circula em torno do BGP actualmente. E, apesar destes valores, em [Hus02a, Hus02b] o autor assume não existir motivos para preocupação imediata, ou em um futuro próximo. Mas o problema encontra-se no futuro não próximo onde, eventualmente, teremos que chegar a uma solução que harmonize o núcleo da Internet, acalme as mensagens de actualização do BGP e garanta uma maior estabilidade e convergência em casos de falhas.

2.3.2 Crescimento das tabelas de encaminhamento

Com as tecnologias modernas e, principalmente, com o surgimento dos acessos *multi-homed*, o crescimento das tabelas de encaminhamento ficou super-linear nos meados de 2004, mantendo-se assim até agora. Em Abril de 2010, o tamanho das tabelas de encaminhamento excedia as 310 mil entradas. Em Dezembro de 2011, já excedia em certos casos as 385 mil entradas (Tabela 2.2). E a tendência é para continuar a aumentar.

Em [HA06] foi feito um estudo intensivo dos diversos problemas presentes no BGP, derivados do crescimento da Internet actual. Através da análise dos diversos dados recolhidos, fizeram uma previsão de que, se o ritmo continuasse, o crescimento do número de entradas na tabela de encaminhamento crescia de 176 mil em 2005, para 275 mil em 2008 e finalmente 370 mil em 2010 o que, tendo em conta os valores actuais, de mais de 375 mil entradas, as previsões estavam bastante realistas.

Pelas publicações do site *Potaroo* e do seu criador no seu blogue pessoal [bgp11, Hus02a, Hus02b, as611], temos uma recolha de dados precisa onde se pode verificar que, de facto, existe um grande aumento no número de mensagens de actualização (tanto de anúncio como de supressão), existindo também um grande aumento do número de entradas na tabela de encaminhamento dos *routers* BGP.

Por exemplo, através de medições realizadas no AS 65000 [as611], entre Outubro de 2011 e Dezembro de 2011 obtiveram-se os dados apresentados na Tabela 2.2.

Neste exemplo em particular, o número de entradas nas tabelas de encaminhamento já ultrapassou os 385 mil, isto na FIB, porque se contarmos com todas as entradas que um *router* BGP recebe, então o número sobe para mais de 765 mil. Claro que este número é relativo, já que depende do número de vizinhos que um dado *router* possui.

Se estas tabelas crescerem ao ponto de os *routers* mais antigos não as conseguirem processar convenientemente, estes tornam-se obsoletos, tornando as ligações por eles geridas também obsoletas. Este crescimento faz com que o tempo que leva a estabilizar o

BGP após uma falha seja cada vez maior, já que é necessário processar mais mensagens de actualização, e processar uma tabela cada vez maior. Para tentar contornar este problema, os grandes ASes utilizam uma solução que consiste em não aceitar prefixos com dimensão superior a /24 (entre /25 e /32), o que pode limitar o acesso a certas redes, deixando os serviços pouco fiáveis ou mesmo inacessíveis.

2.3.3 Balanceamento de Carga

Como um correcto balanceamento de carga e uma correcta capacidade de escolher que caminho atravessar até um certo destino é uma das principais preocupações a ter em conta para o desenvolvimento deste trabalho, torna-se importante perceber como é que o sistema actualmente lida com este caso e que medidas permite adoptar para minimizar e/ou contornar este problema.

Quem controla um sistema *multi-homed* acaba por querer tirar partido das características únicas que este possibilita, i.e. ter vários caminhos para fluxo de tráfego. Uma das maneiras de o fazer actualmente é anunciando aos seus provedores mais que um prefixo, um por cada uma das ligações que dispõe já que, ao anunciar apenas um prefixo, as outras ligações poderiam ficar subaproveitadas. Consegue assim dividir a carga entre os diversos caminhos, melhorando a qualidade das ligações. O problema prende-se exactamente com esta divisão. Ao anunciar mais que um prefixo, os seus provedores terão que, em vez de adicionar um prefixo para um cliente, adicionar cada um dos prefixos para o mesmo cliente, tendo como resultado o aumento do tamanho das tabelas de encaminhamento.

Para poder usufruir de um correcto balanceamento de carga, um determinado sistema autónomo necessita de ter o controlo das ligações, tanto do tráfego que entra (*inbound*) como do que que sai (*outbound*), de modo a melhorar a performance global das ligações. A gestão do tráfego que sai é mais simples do que a gestão do tráfego que entra, já que o tráfego que sai é o próprio AS que pode escolher por onde o quer enviar, ao passo que escolher o caminho que o tráfego fará para chegar ao seu AS torna-se uma tarefa mais complicada.

Existem, contudo, métodos para gerir este tráfego que entra, cada uma com as suas vantagens e desvantagens:

Aumento do tamanho do caminho anunciado

Esta técnica consiste em aumentar o tamanho de um determinado caminho anunciado entre ASes, de modo a que esse caminho seja preterido em relação a outros mais pequenos. É feito introduzindo no caminho várias vezes o número do AS que o anuncia. Isto fará o algoritmo de selecção de rotas rejeitar este caminho maior, indo o fluxo pelo outro caminho mais curto. O problema desta técnica é que esta afinação nas rotas tem que ser efectuada depois de conhecido o caminho original, ou seja, é um processo *ad-hoc* que se veio a tornar numa solução utilizada mas pouco fiável a longo prazo.

Em [CL05], desenvolveram um sistema em que não é necessário o processo *ad-hoc*, sondando os diversos canais primeiro e consoante um algoritmo prevêem se o aumento de um caminho anunciado num determinado canal trás, ou não, as vantagens necessárias. Esta solução tem, contudo, diversos problemas, sendo a falta de testes sobre o impacto que teria na Internet real um dos principais *setbacks* desta proposta.

Anúncio Selectivo

Anuncia prefixos que não se intersectam em canais diferentes. Os prefixos são maiores (e.g. em vez de um /16, anuncia dois /17), sendo que o tráfego destinado a estes prefixos chega à rede via os respectivos canais. É simples de implementar mas apenas um ISP é utilizado para cada prefixo, funcionando o outro como alternativa em caso de falha.

Divisão de Prefixos

Divide o prefixo em mais pequenos, mas agora que se sobrepõem (i.e. envia o prefixo original para um canal e.g. /16 e um prefixo mais pequeno para outro e.g. /17), sendo que o prefixo mais especifico (/17) acaba por ser utilizado, controlando assim o fluxo de dados. O canal /17 é utilizado como caminho principal enquanto que o /16 é utilizado como *backup*.

O problema destas duas últimas soluções é que, ao aumentar o número de prefixos, irão aumentar também o tamanho das tabelas de encaminhamento do BGP. É por causa deste facto que os *routers* de BGP de hoje em dia estão configurados para descartar prefixos anunciados com dimensões superiores a /24 (de /25 a /32).

Abordagens baseadas em *Network Address Translation* (NAT) [Kal] [2]

Esta abordagem é baseada em caixas de NAT para traduzir o endereço de origem num pacote que saia da rede, num endereço externo de um *router* NAT *multi-homed*, de modo a que o tráfego que retorne possa ser afixado ao canal correspondente. Um dos problemas desta solução prende-se com a necessidade do *Domain Name System* (DNS) ter que remover as entradas necessárias aquando a detecção de falhas nos canais.

Rede sobreposta por cima de BGP [ACK03]

Separa a parte das politicas da parte do encaminhamento, ambas suportadas pelo BGP, para que num caminho mais rápido (na rede sobreposta) se possa proceder às mudanças de encaminhamento. As aplicabilidades desta solução passam por melhorar o tempo de falha e balancear o tráfego que entra das redes *multi-homed*. É muito complexa de implementar.

[2] *Network Address Translation* - Técnica que consiste em reescrever os endereços IP de origem de um pacote que passam por um *router* ou *firewall* de maneira a que, por exemplo: Um computador de uma rede interna tenha acesso ao exterior (rede pública).

2.4 Melhoramentos a curto prazo

De forma a atenuar, a curto prazo, alguns dos problemas do BGP, foram criadas algumas medidas, que podem ser interpretadas como melhorias ao BGP.

Estas passam por: tentar prevenir o envio de mensagens de supressão inúteis, já que este facto iria fazer diminuir o tempo de convergência e o número de mensagens a processar pelos *routers*; descartar caminhos obsoletos, descartando rotas da tabela RIB que são afectadas por uma rota que se sabe que falhou; mecanismos de diferenciação das mensagens, onde o objectivo é apenas o de propagar os anúncios dos caminhos óptimos.

O melhoramento mais importante é o de tentar prevenir as mensagens de supressão. Quando um *router* BGP detecta uma falha, envia uma mensagem de supressão para notificar que a ligação falhou e quais os prefixos que ficaram indisponíveis. A recepção de uma destas mensagens leva a que os *routers* tentem encontrar outro caminho para substituir o que acabou de falhar, enviando mensagens de anúncio com o novo caminho encontrado. O MRAI[3] foi introduzido com o propósito de tentar minimizar o impacto desta fase e, apesar de diminuir o número de mensagens por unidade de tempo, acaba por atrasar todo o processo de convergência. O tempo de convergência do BGP a seguir a uma falha é de $n \times MRAI$ [LABJ00], sendo n o tamanho do caminho mais longo (em número de ASes) que um *router* conhece. Uma das soluções avançadas é a de prevenir ao máximo mensagens de supressão, através de, após detectada uma falha, se configurar o caminho com preferência local de valor 0. Isto faz com que a rota que falhou possa ser substituída, dando a possibilidade ao algoritmo de escolha de rotas de escolher outra com preferência local de valor maior que 0.

Estes melhoramentos não são suficientes a longo prazo, sendo que teremos que arranjar um modo de resolver, de forma mais definitiva, os principais problemas inerentes ao BGP.

2.5 Sumário

Neste capítulo damos a conhecer o estado do encaminhamento actual entre os domínios da Internet. Explicamos em que consiste o protocolo de encaminhamento utilizado (BGP), o seu funcionamento e os problemas que apresenta por si só. Depois apresentamos como é que os problemas inerentes ao protocolo se acentuam quando confrontados com a escala da Internet, estando a conduzir a que o tempo de convergência do BGP não pare de aumentar, podendo actualmente atingir mais de uma dezena de minutos. Terminamos o capítulo com alguns melhoramentos a curto prazo que podem ser implementados no protocolo, sendo tentativas de atenuar os diversos problemas apresentados.

[3] *Minimum Route Advertisement Interval* (MRAI): tempo de intervalo mínimo entre o envio de cada mensagem de anúncio. Os valores por omissão para as mensagens em eBGP são de 30 segundos e de 5 segundos em iBGP.

3

Outros modelos de encaminhamento e escolha de caminhos

Para uma melhor definição do âmbito deste trabalho, temos que perceber que alternativas existem actualmente, tanto como substituto directo ao BGP (i.e. arquitecturas de Internet novas), como aquelas que permitem aproveitar o BGP para construir algo que melhore o estado actual do encaminhamento (i.e. que aproveitam, de algum modo, a arquitectura existente). Para além disso, iremos focar-nos em propostas que tenham em comum tentativas de dar à periferia mais responsabilidade na forma de lidar com o problema. Como tal, omitimos referência a outras soluções deste tipo que tentam melhorar o encaminhamento sem a participação da periferia, pois as mesmas não se coadunam com o objectivo do trabalho.

Este capítulo está dividido em três principais temas: na Secção 3.1 apresentamos dois novos esquemas de encaminhamento (i.e. duas novas arquitecturas), o *Feedback Based Routing* e o NIRA; na Secção 3.2 apresentamos uma solução (LISP) que, aproveitando parte da arquitectura actual, constrói uma arquitectura sobreposta, permitindo o uso de vários caminhos, o que se torna interessante para o nosso estudo; por fim, na Secção 2.1 apresentamos um breve resumo do TCP e na Secção 3.3 descrevemos uma solução (MPTCP) que utiliza várias ligações em simultâneo, permitindo solucionar parte dos problemas que nos propomos a atacar.

3.1 Novos esquemas de encaminhamento

3.1.1 Feedback Based Routing

A proposta *Feedback Based Routing* [ZGC03], que é uma abordagem diferente ao encaminhamento entre domínios, separa a informação de encaminhamento nos seus componentes estruturais e dinâmicos. A informação estrutural denota a existência de caminhos e é propagada para a periferia da Internet. A informação dinâmica, denota a qualidade dos caminhos ao longo da Internet. Os *routers* no núcleo da rede (denominados de *routers* de trânsito) apenas propagam informação estrutural, encaminham pacotes segundo a informação de encaminhamento e filtram os pacotes segundo uma lista de controlo de acessos local. Todas as decisões de encaminhamento são feitas na periferia, por *routers* denominados de *routers* de acesso, baseadas na informação estrutural que receberam dos *routers* de trânsito e em medições de desempenho ponto-a-ponto.

A proposta assume que a Internet é modelada como um grafo directo, consistindo por um nó por cada AS e rede periférica. Os canais entre vértices representam uma relação entre ASes, onde cada canal pode representar um caminho de rede ou muitos caminhos físicos.

A informação estrutural é propagada pelos *routers* de trânsito, que associam a cada canal a que estão ligados uma marca temporal e um prazo, que não deve ser inferior a uma hora. Um canal é removido do grafo se o prazo chegar ao fim, ou seja, se não for refrescado. Os *routers* anunciam periodicamente a existência dos caminhos mas a perca deles não é explicitamente anunciada ou seja, apesar de ser em contextos diferentes (i.e. o BGP com caminhos e o *Feedback Based Routing* com canais) não existe, neste sistema, o equivalente às mensagens de supressão.

Após receberem informação estrutural, os *routers* de acesso constroem o grafo da Internet. Para cada destino tentam encontrar dois caminhos, que devem ser o mais dispares possível isto é, sem canais comuns. Se tanto a origem como o destino forem *multi-homed*, as duas rotas poderão ser disjuntas nos vértices intermédios. Uma rota será utilizada como primária e a outra como *backup*.

Os *routers* de acesso são responsáveis por inserir a informação de encaminhamento (denominada *Internet Relay Token* - IRT) nos pacotes IP. Um IRT contem a lista dos ASes pelos quais um pacote passa. É um encaminhamento do tipo *source routing*[1], agora com a granularidade dos sistemas autónomos. Utiliza como principal protocolo de transporte o TCP para disseminar a informação de encaminhamento. Quando o *router* de acesso inicia, escolhe o caminho com menor saltos entre ASes como sendo a rota primária. Nas operações seguintes, o custo de um caminho é o RTT para o destino, escolhendo o caminho com menor custo como rota primária. Continua a computar periodicamente a rota primária e de *backup* baseando-se na visão actual que tem da rede. Jogando com os RTTs

[1] *Source Routing* - Técnica onde quem envia um pacote pode especificar o caminho que o pacote irá atravessar na rede, através da adição, ao cabeçalho do pacote IP, dos diversos endereços a atravessar.

e tendo em conta as rotas actuais, mantém os dois caminhos o mais actualizados possível.

Em comparação com o BGP, a computação das rotas, em caso de falhas, pode ser adiada mais tempo pelo facto de existirem duas rotas possíveis de utilizar. A probabilidade das duas falharem em simultâneo, tendo em conta que são, tanto quanto possível, rotas disjuntas, é menor. Contudo, assumir que ligações disjuntas falham de forma disjunta pode não ser totalmente verdade, já que podem existir dependências entre os diversos ASes que não se conhecem ou mesmo que ambas as ligações acabem por utilizar um mesmo canal físico, sendo que se este falhar, ambas as rotas falhariam. A quantidade de mensagens trocadas neste sistema também é muito menor, devido à separação da informação estrutural e dinâmica. Os requisitos dos *routers* de trânsito são substancialmente reduzidos, já que os seus recursos não estão directamente relacionados com o tamanho da Internet, ou seja, são independentes do número de ASes e de prefixos existentes. No BGP, os *routers* têm todos que computar os caminhos e efectuar actualizações às tabelas de encaminhamento. Neste sistema apenas os *routers* de acesso têm que fazer estes cálculos, deixando o núcleo da Internet mais independente da sua própria expansão.

O problema desta proposta prende-se principalmente com o uso de *source routing*. Um dos problemas do *source routing* é que este método permite ataques ou visualizações de informação indesejadas e requer cabeçalhos de dimensão variável. Com efeito, pode-se através de *source routing* conseguir aceder a um computador que está por trás de uma rede privada, se conhecermos o IP de outra máquina que esteja nessa rede, também ligada à Internet. Outro problema é o de introduzir uma carga extra no encaminhamento de cada pacote, devido ao *overhead* de comutar entre os diversos endereços IP presentes no pacote (tendo de alterar apontadores e valores no cabeçalho). Actualmente, qualquer pacote que utilize este tipo de encaminhamento é muitas vezes ignorado pelos ISPs. Dai que o uso de *source routing* seja o principal factor que torne o *Feedback Based Routing* em algo que dificilmente seria aplicável na Internet que conhecemos.

3.1.2 NIRA

O NIRA [YCB07] é uma proposta de uma nova arquitectura que dá a hipótese ao utilizador de escolher rotas a nível de domínio (escolher quais ASes um pacote irá atravessar).

Os provedores de serviço de topo, ou seja os que não compram serviços de trânsito de nenhum outro provedor, são catalogados como sendo *tier-1*. Todos os *tier-1* estão interligados, formando o núcleo da Internet. Um dado computador consegue visualizar uma região limitada da Internet, que consiste no(s) seu(s) provedor(es), no(s) provedor(es) do(s) provedor(es) e por ai adiante, até se atingir um *tier-1*. Também se incluem os provedores que tiverem relações de *peering* com o(s) seu(s) provedor(es). Esta região é vista como um grafo ascendente, sendo descoberta através de um protocolo denominado *Topology Information Propagation Protocol* (TIPP).

O TIPP corre entre *routers* das periferias dos provedores e fora do núcleo da Internet (os *tier-1* utilizam algoritmos escolhidos entre eles, e.g. BGP). Tem dois componentes: um

path-vector e um *link-state*.

path-vector
: Utilizado para distribuir as rotas no grafo ascendente ou seja, informação de endereçamento, onde se informa um utilizador dos seus provedores directos e indirectos, assim como das rotas de trânsito entre esses provedores (relações de *peering*). Este componente do TIPP não selecciona caminhos, apenas mostra os caminhos que se pode tomar.

link-state
: Componente baseado em políticas para informar o utilizador das mudanças dinâmicas na rede, e.g. um domínio pode escolher o que enviar a um vizinho à granularidade do canal (escolher ligação a ligação).

O modo de representar uma rota no NIRA é utilizando um endereço hierárquico com raiz no provedor (*provider-rooted*), que mostra um segmento da rota entre o utilizador e um provedor do núcleo. Ou seja, o endereço que um determinado *tier-1* fornece a cada um dos provedores clientes é uma subdivisão do seu prefixo. Por sua vez, esses provedores alocam aos seus clientes uma subdivisão do seu prefixo, repetindo o processo até se chegar ao utilizador final. Assim, o endereço IP de um utilizador final pode ser seguido até um certo *tier-1*, como se de um grafo ascendente se tratasse. Para representar os endereços IP utilizam IPv6, devido ao seu maior espaço de endereçamento. Utilizam 96 bits para designar prefixos entre provedores e clientes e 32 bits para designar endereços dentro de cada um desses prefixos. Criam também uma gama de prefixos para ligações entre *peers* que, como estas ligações estão directamente relacionadas entre eles, um pacote não necessitaria de atravessar o núcleo da rede.

Para um computador comunicar com outro, necessita saber que rotas pode utilizar. Para isso, propõem um serviço de mapeamento (semelhante em alguns aspectos ao DNS), onde se mapeia o nome de um destino nos segmentos das rotas (nos endereços) que este pode utilizar. Este sistema tem o nome de NRLS (*Name-to-Route Lookup Service*). Combinando a informação aprendida através do TIPP com a informação que adquire do NRLS, consegue-se escolher um endereço de origem e um de destino, mapeando tanto a árvore ascendente (do cliente de origem para o provedor) como a árvore descendente (do provedor para o cliente de destino).

Nesta proposta existe uma excepção ao encaminhamento. Nos casos em que existe troca de informação entre dois nós através de ligações de *peering*, o mecanismo normalmente utilizado de disseminação de mensagens pelos grafos (ascendente e descendente) não funciona correctamente, sendo necessário recorrer à utilização de *source routing* na ligação entre os dois *peers*. Contudo, é algo que não acontece com frequência já que, por norma, estas ligações funcionam apenas em casos de suporte a ligações que falharam e.g. se A tem ligação por *peering* a B, e o provedor de B ficar indisponível, A encaminha o tráfego de B momentaneamente através do seu provedor.

Este sistema tem então uma vantagem em relação ao *Feedback Based Routing* (Subsecção 3.1.1) já que, tirando a excepção descrita anteriormente, não requer a utilização de *source routing*. Tem também outras vantagens, como: permitir a escolha dos caminhos a percorrer ao utilizador final; aliviar a carga sobre o núcleo da rede; permitir reduzir o número de entradas nas tabelas de encaminhamento do núcleo, já que estas estarão confinadas aos vizinhos, pelo que se espera que sejam em número muito inferior aos das tabelas de encaminhamento do BGP. A principal desvantagem nesta proposta é o facto de que o número de endereços que cada cliente final possui pode ser significativo. Esta situação tornaria a escolha de caminhos em algo pesado para aos utilizadores, já que nesta proposta toda a complexidade de selecção de rotas foi trazida para a periferia.

O modelo de uso comum do NIRA prende-se com o uso de agentes, onde um software especifico escolhe as rotas baseando-se em preferências dos utilizadores. A escolha, porém, não está confinada aos utilizadores, sendo que os próprios domínios podem utilizar mecanismos para escolherem rotas pelos utilizadores. Por exemplo: se um domínio utilizar uma caixa NAT, esta pode seleccionar as rotas de acordo com os computadores no domínio.

3.2 *Locator Identifier Separation Protocol*

Na Internet, as redes operam actualmente no mesmo espaço de encaminhamento e endereçamento, combinando duas funcionalidades: *Routing Locator*s (RLOCs), que descrevem como um dispositivo está ligado à rede e os *EndPoint Identifier*s (EIDs), que definem quem somos na rede, ou seja um dado endereço IP. A separação destas duas funcionalidades (denominada *Locator/ID split*, ou separação localizador/identificador) tornou-se um dos maiores objectivos na construção de uma nova arquitectura da Internet [JMY+08].

O crescimento da periferia afecta directamente o tamanho das tabelas de encaminhamento do núcleo, sendo que qualquer rede periférica instável pode inundar toda a Internet com mensagens de actualização. A ideia de separar a localidade do identificador ataca directamente este problema, na medida em que consegue separar os prefixos das redes periféricas do tráfego no núcleo. Esta separação aumenta a escalabilidade do sistema de encaminhamento permitindo maiores agregações de RLOCs, uma identificação persistente no espaço de EIDs e, em alguns casos, aumentando a segurança e eficiência da mobilidade na rede.

Esta separação pode ser feita ao nível do utilizador final, e.g. [Al-06, ABH10, Ste07], ou de uma determinada rede, sendo uma das soluções propostas o LISP [Hus06, lis04a, lis04b, lis04c, brk04].

Este protocolo está desenhado para ser um protocolo de *map-and-encap*[2] que implementa a separação dos endereços Internet em EIDs e RLOCs.

[2]Um protocolo de *map-and-encap* (mapear e encapsular), mapeia o EID de destino no RLOC onde esse destino se encontra. Depois de mapear, encapsula o pacote com o seu RLOC como endereço de origem e envia para o RLOC mapeado no passo anterior.

Esta proposta não necessita de mudanças drásticas nas estruturas já existentes, funcionando como uma camada intermédia na arquitectura da Internet (entre a camada *data-link* e a camada IP). Como tal, torna-se independente da família de endereços utilizada, pelo que pode funcionar tanto com IPv4 como com IPv6.

O LISP propõe-se a melhorar o *multi-homing* e a reduzir o tamanho e o dinamismo das tabelas de encaminhamento do núcleo da Internet.

O modo de funcionamento do LISP está directamente relacionado com a utilização de túneis. O mecanismo de túneis permite construir redes sobrepostas que alcançam melhores resultados (e.g. em termos de desempenho) face às redes físicas por norma utilizadas. Para este fim, utilizam IP sobre *User Datagram Protocol* (UDP), com a adição de um cabeçalho LISP especifico entre o cabeçalho exterior de UDP e o cabeçalho IP interior original (i.e. com os EIDs de origem e destino). No cabeçalho exterior encontram-se os RLOCs de origem e destino utilizados no encaminhamento LISP. Na Figura 3.1 podemos ver um exemplo de um pacote LISP em formato IPv4.

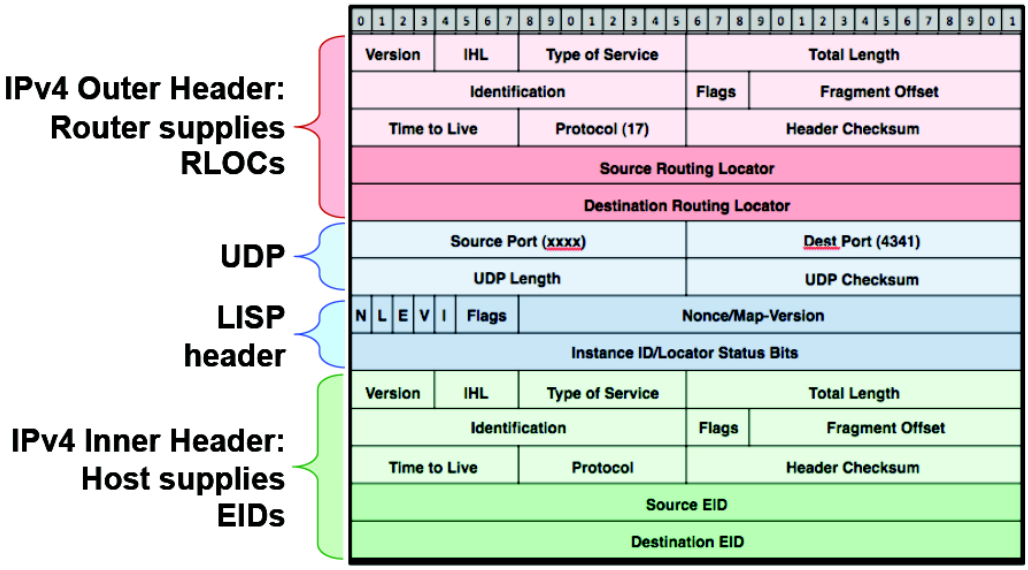

Figura 3.1: Exemplo de um pacote LISP em formato IPv4 [brk04]

Esta alternativa assume dois elementos principais de rede e um terceiro genérico:

***Egress Tunel Routers* (ETR)**
: Responsável por receber uma mensagem LISP encapsulada e, caso seja ele o destinatário, extrai o pacote interno e envia o mesmo para a sua rede interna;

***Ingress Tunnel Routers* (ITR)**
: Responsável por encapsular uma mensagem recebida por um dos computadores da sua rede numa mensagem LISP, enviando-a em seguida para a Internet;

xTR
: Este terceiro elemento designa um *router* LISP que possui as duas funcionalidades activas, *Ingress* e *Egress*.

3.2.1 Mapeamento de EIDs em RLOCs

Como esta alternativa se baseia num protocolo de mapear e encapsular, uma das operações críticas é a forma como o sistema de mapeamento de EIDs em RLOCs está implementado.

A proposta assume dois tipos de interacção para suportar o mapeamento de EIDs em RLOCs:

Pedido de mapeamento (*Map Request*)
: Um ITR pode inquirir o sistema de mapeamento enviando este tipo de pacote para pedir um mapeamento especifico.

Resposta ao mapeamento (*Map Reply*)
: Enviado por um ETR como resposta aos pedidos de mapeamento.

Para introduzir o sistema de mapeamento foram propostas várias soluções. A mais significativa, e que foi a escolhida para a implementação actual, foi a *Alteranative-Topology* (ALT), constituindo assim o LISP-ALT [MFFL]. É uma camada lógica para manusear o mapeamento, que tira partido do funcionamento do BGP e de túneis GRE[3], de modo a construir uma rede sobreposta de dispositivos que apenas anunciam prefixos EID. Utiliza o BGP para propagar informação de alcance dos prefixos EID utilizados pelos xTR, de modo a ser possível encaminhar os diversos tipos de pacotes de mapeamento.

As mensagens de mapeamento do LISP são enviadas e recebidas por UDP, sendo que as mensagens de pedido de mapeamento são enviadas através do sistema LISP-ALT, mas as mensagens de resposta são enviadas pela rede normal.

O mapeamento possibilita ainda definir atributos de prioridade e de peso nos diversos caminhos. A prioridade diz ao ITR para qual dos RLOCs de um determinado destino (i.e. para qual ETR) é que deve enviar as mensagens. O peso permite a um ITR dividir a carga entre RLOCs/ETRs de prioridades iguais (i.e. é uma percentagem do tráfego que deve ir para cada ETR). Estas propriedades são úteis em casos de *multi-homing*, já que permitem escolher os pesos das rotas e permitir a utilização de várias rotas em simultâneo.

De modo a efectuar correctamente o mapeamento e, por consequência, o encaminhamento, cada xTR acede a duas estruturas: uma base de dados e um sistema de cache.

A base de dados possui o mapeamento entre EIDs e RLOCs locais ao xTR (dentro do domínio). O seu principal propósito é o de auxiliar um *router* a perceber se o destino se encontra na própria rede ou numa rede exterior. O tamanho desta base de dados é directamente proporcional ao número de EIDs e xTRs que o domínio possui. Como este número é finito, a base de dados não constitui um problema de escalabilidade no sistema.

A *cache* guarda temporariamente os mapeamentos para os prefixos EID que não fazem parte do domínio local. Isto é necessário para encapsular correctamente os pacotes que

[3] *Generic Routing Encapsulation* (GRE), é um protocolo para encapsular pacotes IP sobre IP. Utiliza túneis para enviar um pacote IP de uma rede para outra, sem que esse pacote seja tratado ou filtrado pelos *routers* intervenientes como sendo um pacote IP.

saem, em particular para escolher o RLOC a ser utilizado como destino no cabeçalho exterior do pacote, de uma forma mais rápida e eficaz. O tamanho médio da cache não cresce linearmente com o número de sistemas finais, já que esta guarda apenas os prefixos para os EIDs recentemente utilizados/em utilização, o que atribui à cache do LISP boas propriedades de escalabilidade.

3.2.2 Funcionamento

Quando um computador num domínio LISP emite um pacote IP, utiliza o seu EID como endereço de origem e um EID de destino como endereço de destino no cabeçalho do pacote. O EID de destino pode ser adquirido através de um servidor de nomes, como o caso do DNS (Figura 3.2 - passo 1). Nesta proposta assumem que os EIDs são encaminhados dentro de algum escopo (e.g. ao nível do domínio onde se encontra). Quando o pacote chegar a um ITR (Figura 3.2 - passo 2), indica que o destino está noutro domínio e o ITR irá ter que encapsular o pacote. Se o ITR possuir o mapeamento do destino em cache (Figura 3.2 - passo 3), insere, no cabeçalho do pacote encapsulado, o RLOC de destino como sendo o endereço de destino e o seu RLOC como sendo o endereço de origem. Se não possui o mapeamento em cache, envia um pedido de mapeamento por exemplo, através da LISP-ALT. O pedido é encaminhado até chegar ao ETR de destino. Este verifica se é o destinatário e, em caso positivo responde com o tipo de pacote de resposta ao mapeamento. Esta resposta irá indicar ao ITR que originou a mensagem que pode enviar as próximas mensagens agora pela rede normal, sem ser necessário o uso da LISP-ALT, pois já possui em cache o mapeamento para o destino. Ao receber um pacote LISP (Figura 3.2 - passo 7), o ETR verifica na base de dados local se o EID destino pertence à sua rede. Em caso afirmativo, desencapsula o pacote, enviando-o para o destinatário (Figura 3.2 - passo 8).

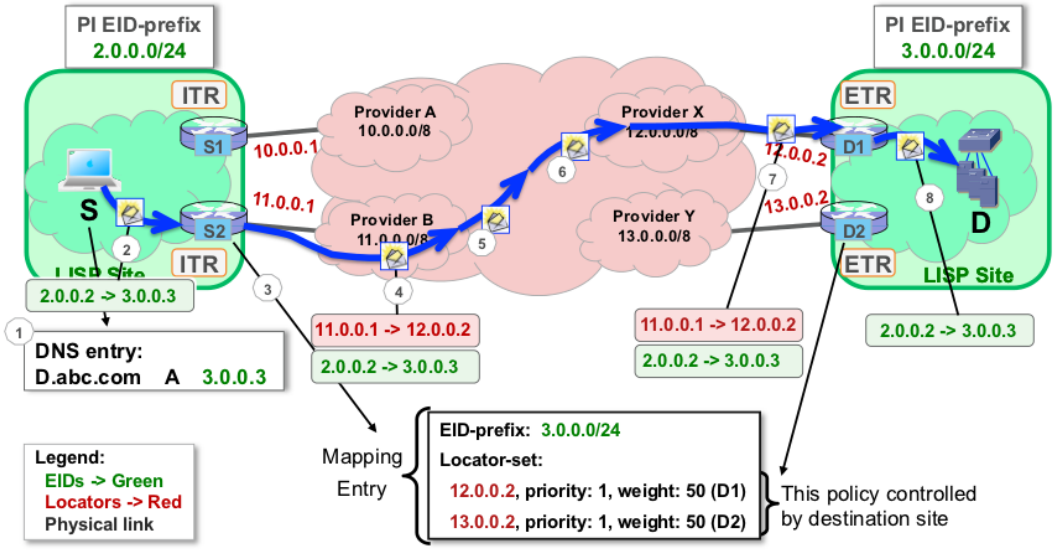

Figura 3.2: Exemplo do funcionamento de LISP em Unicast [brk04]

Os mapeamentos são guardados em cache durante um certo tempo, sendo posteriormente descartados. Como as entradas na cache expiram, é simples perceber que são inseridas a pedido (*on-demand*), apenas quando são necessárias, sendo que um *timeout* adequado é a chave para um bom desempenho na cache. Isto torna a cache um sistema crítico pois o seu conteúdo, tamanho e eficiência está directamente relacionado com o volume de tráfego que flui num determinado xTR. Em particular, o primeiro pacote destinado a um EID que não está em cache, irá gerar um *cache-miss*. Este *cache-miss* gera uma mensagem do ITR ao sistema de mapeamento e o pacote que a gerou acaba por não poder ser encapsulado, já que não existia mapeamento para ele, sendo descartado. Os pacotes seguintes já poderão ser enviados normalmente, pois o mapeamento para os RLOCs destino já existe em cache.

3.2.3 Vantagens e Desvantagens

Como todas as propostas, também o LISP tem os seus prós e contras.

A principal vantagem do LISP, derivado da separação da localização e identificação, é o de permitir reduzir o tamanho das tabelas de encaminhamento no núcleo da rede [QIDLB07]. Consegue-se assim um encaminhamento BGP mais leve, movendo a complexidade do encaminhamento do núcleo para a periferia.

Outras vantagens a assinalar neste sistema:

- a melhoria no que toca à mobilidade, já que um sistema não está com um IP "preso"à rede onde se encontra. Com um EID único por sistema/site/organização consegue-se mover sistemas inteiros de uma rede para outra, sem que estes necessitem sofrer uma total renovação;
- devido ao sistema de prioridade e peso presentes no sistema de mapeamento, esta proposta permite a configuração mais simples e eficaz de sistemas multi-homed;
- devido à independência da família de endereços, consegue suportar comunicações entre IPv4 e IPv6;
- como é implementado sob um sistema de túneis, conseguem criar mecanismos para possibilitar a comunicação entre sistemas LISP e sistemas não LISP, nomeadamente através de mecanismos como caixas NAT (LISP-NAT) ou *routers* que actuam como *proxies* entre a rede LISP e não LISP;
- apesar de ser a Cisco que incentiva a maioria do desenvolvimento do LISP, esta proposta é do âmbito do IETF [iet] e é de código aberto, ou seja, qualquer pessoa/instituição pode participar activamente no seu desenvolvimento, o que tem contribuído para o rápido crescimento desta alternativa.

Como desvantagem, o facto do sistema de mapeamento implementado (i.e. LISP-ALT) continuar a utilizar o BGP faz com que não se revolva o problema do melhor caminho no encaminhamento. Continuamos a não garantir que um caminho considerado como mais curto pelo BGP seja, de facto, o melhor caminho. Contudo, este problema é

atenuado nesta proposta, já que o peso que o BGP tem que suportar é consideravelmente menor que na Internet actual.

Apesar do LISP possuir alguma segurança e.g. encriptação SHA-1 entre os ETRs e o serviço de registo de mapeamento ou a autenticação entre os peers no ALT-BGP, não possui segurança contra ataques de *man-in-the-middle*[4].

Esta proposta tem ainda em conta questões de performance: no que toca ao tamanho dos pacotes após a adição dos cabeçalhos extra, temem que possam exceder o *Maximum Transmission Unit* (MTU); na perca de pacotes e latências de pesquisa no sistema de mapeamento e no retorno a pedidos de mapeamento por parte dos ETRs.

3.2.3.1 Acessibilidade em caso de falhas

As vantagens na flexibilidade e no controlo fornecidos por esta proposta vêm com um custo associado, o aumento da complexidade de detecção de acessibilidade. Ao efectuar o mapeamento, um ITR tem que saber se o ETR de destino se encontra disponível, tendo que recuperar por ele próprio caso detecte alguma falha, e.g. escolhendo outro ETR possível para o mesmo destino. Um modo simples de efectuar esta verificação seria quando um ITR recebesse uma mensagem de ICMP[5] de destino inacessível. O problema é que as mensagens ICMP são muitas vezes limitadas ou mesmo não enviadas. Para além disso, o ICMP é propício a ataques de *spoofing*. Para um xTR verificar se um dado ETR está acessível, foram definidos vários mecanismos [FFM+11, Sau11]:

Locator Status bits

Através dos 32 bits de estado presentes no cabeçalho LISP, i.e. *Instance ID/Locator Status Bits* (ISB) (Figura 3.1), um xTR consegue verificar quais ETRs num determinado destino é que estão disponíveis. Quando um ITR encapsula um pacote, associa, para cada ETR na sua rede local, uma posição do campo ISB com o valor 1. Ao receber uma mensagem, um ETR verifica cada um dos valores do ISB em busca de mudanças. Quando detecta que a posição correspondente a um determinado RLOC/ETR, associado ao EID que enviou a mensagem, mudou de valor de 1 para 0, então sabe que esse RLOC ficou indisponível. A partir dai tentará ao máximo, quando actuar como ITR, não enviar mensagens para o RLOC em baixo, resumindo o envio para este apenas se receber uma mensagem com a posição novamente a 1.

Algoritmo *Echo Nonce*

Esta técnica consiste em utilizar os bits N (*nonce*) e E (*echo*) presentes no cabeçalho LISP do pacote LISP (Figura 3.1). Um ITR, ao enviar uma mensagem, faz *set* destes dois bits e atribui ao campo *Nonce/Map-Version* um número aleatório. Quando o ETR recebe a mensagem encaminha para o destino na sua rede normalmente. Contudo, da próxima vez que o ETR (agora em modo ITR) enviar uma mensagem de

[4]Técnica onde alguém "escuta"o canal, no meio da ligação, de modo a aceder a informação sem os intervenientes em qualquer das extremidades notarem

[5]Internet Control Message Protocol (ICMTP) - protocolo integrado no IP, utilizado para fornecer relatórios de erros à origem

volta para o ITR (agora em modo ETR), irá incluir novamente o número aleatório e o bit N preenchido, limpando o bit E. O ITR (agora ETR) ao receber o pacote compara o *nonce* e verifica que somente o bit N está preenchido, reconhecendo que o caminho de si para quem enviou a mensagem está disponível. Este mecanismo não resolve o problema da acessibilidade por completo, já que o xTR que recebeu uma mensagem, pode não ser o mesmo xTR dessa rede que irá enviar o *echo* de volta.

Algoritmo de sonda de RLOCs (*RLOC Probing*)

Este mecanismo consiste em sondar activamente todos os RLOCs que um certo ITR possui mapeados em cache. Para isso utilizam um bit no pacote de dados de pedido de mapeamento e de resposta ao mapeamento. Um pedido de mapeamento utilizado como uma sonda de RLOC não é encapsulado nem atravessa o sistema de mapeamento (e.g. LISP-ALT). Quando um ETR recebe um pedido de mapeamento com o bit de sonda activo, envia para o ITR de origem uma resposta ao mapeamento com o bit de sonda também activo. Desta forma quando o ITR (agora ETR) receber a mensagem, sabe que o destino existe e está disponível. Também se pode utilizar este pedido de mapeamento especial, como forma de actualizar a cache de um certo ITR. O ETR quando recebe a mensagem, pode enviar uma resposta ao mapeamento que contenha os dados actuais dos mapeamentos EIDs/RLOCs na sua rede. Apesar de ser uma técnica que produz bons resultados em termos de acessibilidade, tem um grande peso na performance já que envia mensagens para todos os RLOCs mapeados em cache.

A vantagem destas técnicas é que podem ser utilizadas em simultâneo, conforme forem necessárias, e dependendo do contexto que se pretende testar a acessibilidade.

3.2.4 LISP Mobile Node

O *LISP Mobile Node* (LISP-MN) [FLMW11] é uma das muitas formas de aplicabilidade do LISP. Surgiu da necessidade de providenciar a um sistema móvel a capacidade de mobilidade sem a perda de conexão. Apesar de ainda se encontrar em desenvolvimento, os objectivos da mobilidade do LISP incluem: permitir que as ligações TCP se mantenham vivas enquanto o dispositivo se desloca, permitindo a comunicação entre diversos nós enquanto este movimento ocorre; permitir o uso de *multi-homing* nos dispositivos (i.e. utilizar múltiplas interfaces de forma concorrente); permitir que qualquer nó móvel da rede possa fazer de servidor perante outro qualquer nó; providenciar caminhos de dados bidireccionais.

Cada LISP-MN necessita de utilizar um serviço de Mapeamento (e.g. LISP-ALT), assim como a tecnologia LISP *Interworking* [LFFM11], de modo a se poder movimentar e ser descoberto de forma eficaz e escalável. Cada nó funciona como sendo um site LISP, funcionando como um ITR e ETR, conforme necessário. Sempre que muda de operadora, um LISP-MN recebe um RLOC, que terá que registar no serviço de mapeamento com o seu EID associado.

Existe uma implementação desta solução, em [CNJ+]. O código é aberto, baseado numa implementação do OpenLISP e LISP-MN.

Esta técnica é uma aplicação interessante do LISP para o desenvolvimento deste trabalho já que permite ver, de uma forma aplicada na realidade, que o LISP permite a escolha e utilização de mais que um caminho para um determinado destino. Permite inclusive que se utilize mais que uma interface, quando tal é possível.

3.3 *MultiPath TCP*

A ideia do MPTCP é a de tirar partido dos caminhos que são ignorados pelo sistema de encaminhamento (e.g. em caso de *multi-homing*), permitindo melhorar a fiabilidade das ligações e aproveitar ao máximo as infraestruturas existentes.

Esta proposta divide-se em dois ramos:

***Multiaddressed MultiPath TCP* [Bon11, SF11]**
: Divide uma sessão TCP em vários fluxos, cada um com o seu endereço de origem e de destino. Pelo menos uma das partes tem que possuir mais que um endereço (de preferência as duas) e ambas têm que implementar MPTCP. Os pacotes são enviados por caminhos diferentes, sendo estes caminhos endereçados pelos endereços distintos que são conhecidos até ao destino. Para criar uma ligação, primeiro estabelecem uma ligação normal com o destino, e somente depois é que tentam estabelecer fluxos adicionais de dados. Consegue manter a compatibilidade com os sistemas intermédios, e.g. caixas NAT.

***One-ended MultiPath TCP* [vB09]**
: Apenas é necessário alterar quem envia a mensagem ou abre a conexão. Este tem que possuir mais que um endereço, independentes do provedor de Internet (*PI address*) onde, se possuir mais que uma interface, uma para cada um desses endereços, pode repartir o envio dos dados pelas ligações associadas. Tira partido dos mecanismos de controlo de congestionamento implementados no emissor i.e. os mecanismos do próprio TCP, pelo que é possível utilizar mais que um caminho desde que o receptor utilize *acks* selectivos (SACK [tcp12]). Também mantém a compatibilidade com sistemas intermédios.

Como cada fluxo comporta-se como sendo uma ligação TCP comum, possui o seu próprio espaço de sequência dos pacotes, assim como as suas próprias janelas de congestionamento. Consegue assim adaptar cada fluxo às diversas condições que encontra ao longo de um caminho. Para além disso, mantém um número de sequência e janela de congestionamento global [RHF09], que permite juntar a informação de forma correcta nas junções dos fluxos, permitindo por exemplo, fechar toda a ligação (e respectivos fluxos) caso seja necessário. Aproveitando as dinâmicas do controlo de congestionamento nos diversos fluxos, o MPTCP consegue mover, de forma explicita, o tráfego de caminhos mais congestionados para outros menos congestionados. O uso de diversos fluxos

em simultâneo indica que, se um deles falhar, todo o tráfego é movido para os restantes fluxos, sendo que qualquer pacote perdido é retransmitido.

O MPTCP é uma extensão ao TCP, desenhado para ser compatível com as aplicações existentes, dando origem a duas classes distintas: a primeira, são as aplicações que já existem, que necessitam de continuar a funcionar sem serem modificadas, e que utilizam a API por omissão de Sockets, sendo o MPTCP transparente para estas aplicações; a segunda classe são aplicações cientes de MPTCP, sendo que utilizam extensões à API por omissão, de modo a tirarem melhor partido do MPTCP.

Em [SF11] desenvolveram uma API que possibilita criar aplicações que explicitamente tiram partido do MPTCP. Esta API permite chamar funcionalidades especificas, funcionando como uma extensão à API de Sockets. Os quatro principais fundamentos desta API são: uma aplicação deve ligar e desligar o MPTCP sempre que for necessário (i.e. comutar entre TCP *single-path* e *multipath*); uma aplicação deve poder restringir que endereços é que podem ser utilizados pelo MPTCP; por fim, uma aplicação deve conseguir obter informação sobre cada um dos fluxos MPTCP.

Existe uma grande preocupação em fazer com que a performance do MPTCP não seja pior que a do TCP normal (i.e. *single-path*) [SF11]. Ao nível de uma aplicação, o ganho é significativo já que, a utilização de múltiplos caminhos, faz aumentar o *throughput* global da ligação. Como o MPTCP permite adaptar o fluxo dos caminhos que utiliza, proporcionando *fairness* entre ligações *single-path* e *multipath*, garante que nenhuma aplicação tem uma ligação com performance pior que o TCP (*single-path*).

Com as vantagens que esta proposta apresenta, i.e. utilização de múltiplos caminhos; nível de controlo e segurança presentes no TCP *single-path*; suporte às aplicações já existentes; uma API para desenvolver aplicações viradas especificamente para o MPTCP, o MPTCP tem todas as bases para se tornar numa proposta viável para a Internet, resolvendo parte dos problemas de congestionamento existentes. O problema mais notável está no facto de que, caso se queira utilizar MPTCP, necessitar mudanças em ambos os extremos da comunicação, o que acaba por tornar a sua aceitação num assunto mais delicado.

É, no entanto, uma das propostas a ter em conta já que permite, dentro de um certo nível, a escolha ou pelo menos a utilização de vários caminhos, e o suporte a nível aplicação, o que se revela um factor de interesse para o desenrolar deste trabalho.

3.4 Sumário

Neste capítulo passámos em revista várias alternativas ao encaminhamento actual que tiram partido da maior inteligência na periferia, passando por arquitecturas totalmente novas (*Feedback Based Routing* e NIRA), a um misto que aproveita parte do encaminhamento actual, renovando a arquitectura da Internet, com o recurso a túneis e à separação entre EIDs e RLOCs (LISP). Continuamos as alternativas referindo uma última que permite utilizar vários caminhos em simultâneo (*MultiPath TCP*), e que se revelou uma fonte

de inspiração para a nossa proposta. Estas alternativas serviram-nos em parte de inspiração e permitem julgar de forma mais clara a proposta, assim como as suas vantagens e as suas limitações.

4

Proposta

A fiabilidade e o desempenho das redes de computadores e das aplicações que as mesmas suportam repousam em grande medida na utilização generalizada de redundância, quer ao nível rede, quer ao nível aplicacional. Geralmente as redes são desenhadas de modo a existirem diversos caminhos entre as principais regiões das mesmas, ou pelo menos um nível de diversidade adequado nos caminhos mais críticos. Ao nível aplicacional, quanto mais crítico é o sistema distribuído, maior é o nível de redundância das suas componentes. Por exemplo, o DNS baseia-se na existência de pelo menos 2 servidores autoritários para cada domínio, mas para além dos 13 *root name servers* [Kar] oficiais, existem hoje em dia várias dezenas de *root name servers* acessíveis por *anycasting* [Mee].

As redes estão, regra geral, estruturadas de tal modo que a redundância está confinada a cada camada, não sendo a diversidade da mesma visível entre camadas. Ao nível rede a diversidade dos caminhos é explorada de diversas maneiras: os protocolos de encaminhamento mais comuns alteram os caminhos usados sempre que há falhas; alguns protocolos de encaminhamento permitem distribuição de carga automática entre caminhos de igual custo; as técnicas de engenharia de tráfego optimizam a utilização de rede distribuindo os diferentes fluxos de pacotes através dos diferentes caminhos disponíveis. No entanto, a interface de rede IP não permite aos níveis superiores explorarem essa diversidade, pois a sua exploração está-lhes, no essencial, vedada.

A grande maioria dos serviços críticos acessíveis via *Hypertext Transfer Protocol* (HTTP) [RUJ+99] estão replicados por diversos servidores. Esta replicação pode ou não ser visível pelos clientes. No entanto, mesmo quando existem diversos endereços IP que dão acesso ao serviço, os *browsers* HTTP não os exploram, escolhendo apenas um deles. Geralmente, a redundância existente apenas é explorada dentro do serviço, sendo a mesma

transparente para os seus clientes[1].

Existem pelo menos dois argumentos a favor de a diversidade ser exposta às camadas superiores ou aos clientes: primeiro, se for mais fácil mascarar as falhas ao nível superior ou nos clientes que ao nível inferior ou no serviço; segundo, se for mais flexível optimizar e explorar alternativas ao nível superior ou pelos clientes. Apesar de tradicionalmente o nível rede reservar para si todas as responsabilidades em termos de como optimizar e mascarar falhas, procurando tornar transparentes tais funcionalidades ao nível transporte e aplicação, existem diversos argumentos em favor de colocar em questão essa aproximação.

Muitos *hosts* têm hoje em dia diversas interfaces que providenciam diversas alternativas para chegar a um destino. Como referido no capítulo anterior, estudos como o NIRA (Subsecção 3.1.2) propõem que os *hosts* tenham acesso a várias alternativas de encaminhamento através da utilização de diversos endereços. LISP (Secção 3.2) é uma proposta que permite a optimização do encaminhamento na Internet através da exploração do *multi-homing* e diversos endereços. A proposta inicial destinava-se apenas ao encaminhamento *inter-ASes* mas a sua versão denominada mobile LISP (Subsecção 3.2.4) permite a utilização directa da diversidade pelos *hosts*. Finalmente, uma técnica comum para explorar a diversidade na Internet consiste na utilização de um dispositivo, geralmente designado *NAT-box* [nat], ligado a múltiplos ISPs e, portanto, com múltiplos endereços externos. Simultaneamente, inúmeras aplicações *Peer to Peer* (P2P), como por exemplo o *Bittorrent* [Coh], exploram de forma sistemática a diversidade de participantes num enxame (*swarm*) e ganhariam se tivessem uma visão mais clara do custo de comunicar com cada um deles. No entanto, não existem interfaces na rede que exponham essa visão.

A níveis superiores verifica-se que muitas aplicações e sistemas distribuídos, em particular o DNS e as *Content Delivery Network* (CDN) [NSS10], têm um alto grau de redundância e utilizam mecanismos de teste e análise da rede de forma a mascararem as falhas e a dirigirem os clientes para os servidores mais adequados.

Se, por hipótese, um servidor ou um cliente for acessível por vários endereços IP distintos, correspondentes a caminhos distintos para os alcançar, ou se um servidor tiver várias instâncias diferentes, perfeitamente equivalentes, é possível programar os clientes de tal forma que explorem essa diversidade, procurando utilizar o caminho ou o servidor que seja mais eficiente e, no limite, explorem alternativas de caminho ou de servidor quando detectem que as suas escolhas anteriores são menos ineficientes ou até impossíveis devido ao surgimento de uma falha.

Para este efeito, é necessário explorar as diversas alternativas, monitorizá-las, etc. Como isso complica bastante o desenvolvimento das aplicações não é geralmente utilizado. Por exemplo, se o *download* de um objecto se revelar impossível através de uma dada conexão IP, os *browsers* não tentam automaticamente outras alternativas de forma transparente para o utilizador. Existe apenas uma proposta de protocolo que torna esta

[1] O DNS é uma excepção, pois permite que o cliente (geralmente a biblioteca *resolver*) use mais do que um local *name server* para iniciar a sua pesquisa.

exploração fácil ao nível transporte, é o MPTCP (Secção 3.3). No entanto, este protocolo apenas explora a diversidade ao nível rede mas não ao nível aplicação e requer a sua disponibilidade generalizada, o que se advinha difícil pois a introdução de novos protocolos de transporte nunca foi fácil.

4.1 Objectivos da proposta

O objectivo deste trabalho é desenhar e testar um *framework* de monitorização e um padrão de programação que torne mais fácil aos programadores explorarem a diversidade que possa existir ao nível rede, e ao nível aplicacional, de modo a aumentar o desempenho dos serviços e mascarar mais facilmente as falhas que possam surgir.

Para cumprir este objectivo assumimos que um cliente e/ou servidor têm ao seu dispor diversos endereços para um certo destino, ou porque dispõem de várias interfaces, ou porque o nível rede lhes oferece essa possibilidade ou ainda porque têm acesso a servidores distintos que providenciam exactamente o mesmo serviço (e.g. servidores replicados), com idempotência em transacções do tipo pedido/resposta, mesmo quando se usam vários endereços. Este pressuposto prende-se com o facto de que, se uma origem apenas possuir um caminho para chegar a um determinado destino, e este falhar, não há modo de escolher caminhos alternativos. Assim, a ideia é a de desenvolver um padrão de estruturação de interacções pedido/resposta sobre TCP que permita atingir os objectivos acima indicados: mascarar as falhas e escolher a melhor de várias alternativas de caminhos de rede/servidor. Para facilitar as decisões, um *framework* de monitorização apoia as tomadas de decisão ao nível aplicacional sobre o melhor par de endereços origem/destino a utilizar em cada momento para comunicar.

O objectivo é o de, tendo várias alternativas para estabelecer uma conexão TCP entre um cliente e um servidor[2] (ou entre um cliente e servidores equivalentes), mascarar a falha de uma das alternativas passando a usar outra (Equação 4.1) e, se possível, maximizar a velocidade de transferência (Equação 4.2), mas utilizando em cada momento apenas uma das alternativas possíveis. Pretendemos, assim, facilitar o desenvolvimento de aplicações baseadas em TCP que explorem diversos caminhos entre a mesma origem e destino, em que cada uma das alternativas é caracterizada por um par de endereços (IP_{orig}, IP_{dest}). De forma mais rigorosa, o problema pode ser apresentado da forma que se segue.

Dados: H_1 e H_2, dois *hosts*, O o conjunto dos endereços de H_1, D o conjunto dos endereços de H_2. Seja C o conjunto constituído por todos os pares $\{(x, y) : x \in O \wedge y \in D\}$; $v(x, y)$, a função que devolve, em cada momento, a velocidade de transferência que a rede fornece entre H_1 e H_2 via x e y; e M_v o máximo de v em C, pretende-se em cada momento

[2] A mesma visão pode aplicar-se num sistema P2P ao nível de cada uma das interacções possíveis entre dois nós.

seleccionar um par (x, y) tal que:

$$v(x, y) > 0, \text{ e} \tag{4.1}$$

$$M_v - v(x, y) \leq \delta, \text{em que } \delta \text{ é um limiar de optimalidade (e.g. 20\% de } M_v) \tag{4.2}$$

Repare-se que, o par (x, y) que satisfaz as equações acima referidas só existe se, pelo menos um dos caminhos (x, y) permitir o encaminhamento de H_1 para H_2.

4.2 Especificação da proposta

Dos diversos níveis arquitecturais que estão disponíveis, foi seleccionado o nível aplicação, por ser um nível que permite uma solução interessante que possa levar ao desenvolvimento de um elevado número de aplicações que explorem diversos caminhos disponíveis.

A nossa meta principal foi assim a de criar um padrão de programação para o desenvolvimento de aplicações. Um padrão de programação pode ser visto como um *template* para resolver um problema que surge em diversas ocasiões/aplicações [GHJV95]. Desenvolvemos um padrão que permite a construção de aplicações em que a comunicação entre sistemas é feita de forma a que o melhor dos caminhos possíveis seja utilizado durante o maior tempo possível.

Para desenhar e documentar um padrão de programação existem diversos passos a seguir, que vão desde atribuir um nome ao padrão, dizer as suas funcionalidades, o porquê dele existir, que relações possui com outros padrões e que vantagens trará a sua implementação [GHJV95]. Na Tabela 4.1 temos a listagem das características consideradas mais relevantes.

Tabela 4.1: Atributos principais de um padrão segundo [GHJV95]

Atributo do padrão	Descrição	Capítulo / Secção
Nome do padrão	Nome único que identifica o padrão	4.2
Intenção	Descrição geral e o porquê de usar este padrão	4.2
Motivação	Cenário consistindo num problema e num contexto onde este padrão pode ser utilizado	4.2
Aplicabilidade	Situações em que o padrão é usado	4.2.2
Estrutura	Representação gráfica do padrão (diagrama de classes ou sequência)	5.2
Participantes	Lista de classes/objectos utilizados e o seu papel do desenho	5.2
Colaborações	Descrição da interacção entre classes	5.2
Implementação	Implementação do padrão (solução)	5.4
Exemplo de código	Ilustração de como ser utilizado	5.4
Exemplo de uso real	Exemplo de uso real do padrão	5.5
Consequências	Descrição de resultados, efeitos secundários e *tradeoffs* em usar o padrão	6 e 7

Cada uma destas características é explicada neste documento, por vezes em secções ou capítulos diferentes, de acordo com o descrito na coluna "Capítulo/Secção".

4.2.1 Nome, intenção e motivação

Nome do padrão

O padrão per si não possui um nome, mas sim o sistema que o caracteriza. O nome surgiu da necessidade de possuirmos vários caminhos potenciais (*N channels*) para chegar a um determinado destino. Como tiramos partido de um mecanismo de comunicação entre dois pontos, denominado Socket, presente em diversas linguagens de programação, o nome completo para o nosso sistema ficou assim *NChannelSocket*.

Intenção

Proporcionar um conjunto de recomendações, assim como um conjunto de auxiliares de programação para apoiar o desenvolvimento de aplicações que cumpram os objectivos propostos neste trabalho (Secção 4.1).

Motivação

Actualmente não existe, que tenhamos conhecimento, nenhuma *framework* que permita por si só construir aplicações que possibilitem a escolha de caminhos alternativos aquando de uma transferência. Actualmente se estivermos a fazer uma aplicação numa linguagem com suporte ao uso de Sockets, e.g. Java [jave] ou C [GTK+99], que permita a transferência de um ficheiro entre um cliente e um servidor, o uso por si só das bibliotecas nativas a essas linguagens não nos permitem efectuar comutações ou escolhas de canais alternativos para um determinado destino (a não ser, claro, que seja implementado de raiz). Assim, propomos fornecer um meio de fazer aplicações de forma mais simples para o programador, mas que possuam um leque de características superior ao apresentado actualmente pelas respectivas bibliotecas nativas a cada linguagem.

4.2.2 Casos de aplicabilidade do padrão desenvolvido

Em seguida, especificamos onde pode ser aplicado o padrão por nós desenvolvido, i.e. em que contextos se justifica a mudança e a utilização de diversas conexões sucessivas.

A nossa proposta está actualmente pensada para ser utilizada com um modelo *request-reply* (pedido-resposta) simples, i.e. sem pedidos/conexões em paralelo. Um serviço que utilize este modelo tem as seguintes características:

1. um serviço "logicamente unificado" e acessível através de várias vias, nomeadamente, vários pares $IP_origem, IP_destino$;
2. o cliente desse serviço executa várias transacções. Cada transacção requer que haja um resultado e pode ter de obedecer a uma dada ordem pelo que o cliente só pode executar uma transacção de cada vez;

3. para o cliente é indiferente qual/quais dos endereços do serviço unificado utiliza, e também é indiferente qual a interface local para dialogar com o serviço. Pretende apenas seleccionar uma das disponíveis, de preferência aquela em que obtém a melhor qualidade de serviço e progride por transacções *request/reply* ordenadas;
4. o padrão de utilização neste cenário só permite ao cliente usar uma conexão de cada vez, pois há uma ordem a respeitar.

Assim, dos modelos cliente/servidor, serão relevantes os modelos em que:

- exista um servidor replicado ou acessível por vários endereços;
- as interacções entre cliente e servidor forem idempotentes;
- o cliente possua várias interfaces, e.g. vários endereços distintos de rede ou várias interfaces distintas, sendo que no caso de vários endereços distintos de rede o servidor teria que possuir modo de perceber que se tratava de um mesmo cliente, a não ser que a aplicação construída tratasse desse detalhe.

Para além disso, o nosso padrão foi pensado para aplicações que necessitem de transferir um grande volume de dados, e.g. um grande ficheiro/objecto. A razão para considerarmos este caso, prende-se com tipo de conexão existente. Assumimos no nosso trabalho que uma conexão de muito curta ou curta duração, não apresenta vantagens suficientes para ser considerada viável, isto é, a complexidade de inicializar todos os serviços e utilizar as respectivas funcionalidades só é vantajoso se a conexão for de longa/muito longa duração. Tal facto é uma limitação assumida, já que para transferências curtas, e.g. uma simples página HTML, não é tão rentável utilizar o padrão/sistema por nós criado.

Assim sendo, este padrão torna-se vantajoso de ser utilizado quando é pretendido mover objectos com um tamanho considerável (e.g. dezenas de MB), em que se pretende que a conexão seja o mais fiável possível e que, mesmo em caso de falha da conexão, outro caminho alternativo seja utilizado, possibilitando que a transferência continue e não seja terminada de forma abrupta.

O padrão deve ser o menos transparente possível para o programador, nos aspectos fulcrais de mudança de conexão, ou seja, queremos que seja o programador a decidir se quer aplicar ou não as decisões de mudança de ligação e quando o deve fazer. Alertamos contudo que, como explicitado mais à frente, existem modos correctos de efectuar estas decisões e que são os métodos para o qual o padrão está preparado. Qualquer modificação do que é explicado nos capítulos seguintes não é garantido que funcione correctamente.

Para apoiar a utilização do padrão de programação introduzimos também um *framework* de monitorização. Tal *framework* inclui mecanismos que computam um histórico sobre o desempenho de conexões TCP realizadas anteriormente e uma monitorização activa da conexão correntemente utilizada e das alternativas próximas. Com base nestes mecanismos são oferecidos dois serviços de base: uma recomendação sobre se é interessante ou não tentar uma conexão TCP baseada noutro par de endereços origem / destino, e uma recomendação sobre o par de endereços a utilizar na próxima conexão.

Após a decisão do que seria necessário fazer, o próximo passo foi a implementação. Para isso, tivemos que escolher de entre as linguagens de programação conhecidas, em qual é que se justificava investir numa implementação. Cedo apercebemos-nos de que a escolha recairia sobre a linguagem Java, por ser uma linguagem em rápida expansão, e que se revela relativamente simples de aprender e produzir aplicações que funcionam em diversos sistemas (multi-plataforma), sem a aplicação ter que sofrer modificações.

A nossa implementação do padrão depende de um *patch* no *Kernel* (versão 3.0.16) do Linux (mais propriamente na distribuição *Ubuntu* [ubua], versão 11.10) que permite aceder a estatísticas do funcionamento do TCP. Neste sentido, esta versão não é portável na totalidade mas a análise do que pode ser portável neste trabalho é um aspecto que é referido no capítulo do trabalho futuro (Capítulo 7).

4.3 Discussão preliminar das hipóteses subjacentes à proposta

A proposta baseia-se em diversas hipóteses cuja validade depende de vários factores, como por exemplo a generalização de LISP, e cuja discussão ultrapassa o âmbito deste trabalho. No entanto, discutimos em seguida o realismo de algumas dessas hipóteses no momento presente.

4.3.1 O cliente tem diversos endereços

Com a generalização do suporte de LISP ao nível dos *hosts* esta hipótese poderia ser generalizada, no entanto, a mesma já hoje em dia é normal pois, muitos clientes têm diversas interfaces e os sistemas de operação já estabelecem prioridades de utilização das mesmas. Por exemplo, no sistema testado, a interface sem fios só é usada quando a interface com fios não está disponível. Este aspecto foi notado aquando da configuração do sistema para um dos testes efectuados (Subsecção 6.2.6.2), onde para utilizar as duas interfaces (com e sem fios) em simultâneo tivemos que alterar as definições de rede para a ligação sem fios, de modo a esta não utilizar como rota por omissão o que o sistema escolhesse, i.e. a interface por cabo, quando disponível.

Apesar de não ser comum, também é possível um cliente ter várias interfaces virtuais com endereços distintos e dando acesso a caminhos (ou ISPs) distintos na Internet. Isto é fácil de implementar através de um *router* com várias interfaces ligadas a vários ISPs distintos e que através de NAT e endereços de rede distintos na rede interna permitem aos clientes escolher qual o caminho a usar na rede externa.

4.3.2 O servidor tem diversos endereços

As observações sobre LISP e a utilização de vários endereços através de um *router* ligado a diversos ISPs aplicam-se também neste caso. No entanto, esta situação é bem mais comum que a anterior e pode vir a tornar-se mais generalizada se tal se revelar útil. A técnica era mais popular quando a distribuição de carga entre vários servidores

equivalentes se realizava sobretudo através do DNS. O aparecimento de equipamentos especializados na distribuição de carga sobre *pools* de servidores co-localizados diminuiu a frequência da sua utilização. Quando os servidores não são co-localizados a disponibilização de vários endereços é comum, excepto nas CDNs mais sofisticadas que utilizam técnicas de resolução dinâmica de nomes DNS em função da origem dos pedidos dos clientes e da carga dos servidores.

4.3.3 Interacção entre o cliente e o servidor baseada em TCP com várias conexões

Neste caso, a interacção entre o cliente e o servidor é baseada numa sessão suportada em TCP, que dura um período não negligenciável, e que é compatível com a utilização de várias conexões TCP.

Esta situação é bastante comum actualmente na Internet. O exemplo paradigmático ocorre sempre que o cliente necessita de realizar o download de um ou mais objectos imutáveis de um servidor. Tais objectos podem corresponder a recursos informativos clássicos (e.g. ficheiros) mas também podem conter vídeos (*streaming*). No limite, mesmo quando se trata de *streaming live*, a utilização de *streaming* adaptativo sobre TCP [BAB11] é hoje dominante quando o cliente está ligado a uma rede com uma componente final baseada num canal sem fios, como o WIFI ou redes celulares de 3ª e 4ª geração. Nestes casos, a utilização do protocolo HTTP sobre TCP é uma solução comum. Este tipo de cenário representa hoje em dia o tráfego dominante na Internet [GPSH12] e com a generalização das CDNs deixou de ser suportada num único servidor centralizado, mas por um conjunto de servidores distintos mas equivalentes com suporte de idempotência do *download* desses objectos fragmento a fragmento.

Existem igualmente várias outras situações em que são usadas aplicações suportadas em HTTP que, dada a sua natureza, podem suportar replicação em vários servidores e idempotência das transacções pedido/resposta mesmo quando dirigidas a servidores distintos. Um exemplo é o acesso à Internet com base em *proxies*.

4.4 Sumário

Neste capítulo apresentámos o enunciado e os objectivos da proposta de padrão de programação e de *framework* de suporte cuja realização concreta, implementação e teste serão detalhadas nos próximos capítulos.

Dada a crescente necessidade de exploração da diversidade existente na estrutura da Internet e nas aplicações nela suportadas, discutimos a necessidade de se explorarem de forma mais ágil soluções em que essa diversidade deixa de ser transparente e pode passar a ser explorada ao nível transporte e aplicação. Partindo da hipótese de que essa diversidade se manifesta através da disponibilização de diferentes pares de endereços origem/destino, apresentámos a especificação de um padrão de programação e de um

framework de monitorização que permite explorá-la com base na interface de transporte TCP. Finalmente, discutimos brevemente o grau de consistência das hipóteses subjacentes à especificação do padrão no momento actual, tendo concluído que existe um subconjunto de situações, responsável por uma fracção bastante importante do tráfego actual da Internet, em que as hipóteses subjacentes a este trabalho se verificam.

5

Concretização do padrão em Java

Para chegarmos à descrição actual do padrão começámos por examinar a estrutura dos objectos Java Socket, acabando por decidir que o padrão seria aplicado juntamente com uma *framework* que, utilizando o actualmente fornecido pelo Java, iria estender as suas funcionalidades, permitindo implementar as funcionalidades desejadas.

Um Socket não é mais que um canal de comunicação entre dois sistemas, permitindo a transferência de dados entre eles. O sistema de Sockets do Java [javh] faz parte do pacote *java.net* [javf] e é um sistema altamente complexo e com uma vasta hierarquia de classes. Inicialmente estava pensado fazermos de raiz um novo tipo de Socket Java, mas após nos apercebermos da extensão do que actualmente existe, e das suas funcionalidades, optámos por tirar proveito das mesmas, fazendo uma espécie de optimização. O Socket Java actual permite definir um conjunto de características como a capacidade de modificar algumas das variáveis TCP do sistema, i.e. os *timeouts* de leitura e de conexão, e o facto de todo o mecanismo de canais de input e output já existir e estar ao nosso dispor.

Um esquema muito simplista de uma maneira de utilizar Java Sockets está presente na Tabela 5.1 e exemplificado na Listagem 5.1.

Tabela 5.1: Interacção padrão Socket Java

Passo	Descrição
1	Criar Socket, com endereço e porta de destino.
2	Obter canais de input / output do respectivo Socket.
3	Utilizar o canal de output para escrever um pedido para o destino.
4	Utilizar o canal de input para ler a devida resposta.

Listagem 5.1: Interacção padrão Socket Java

```
1  Socket s = new Socket(remote hostname, remote port);
2
3  InputStream is = s.getImputStream();
4  OutputStream os = s.getOutputStream();
5
6  os.write("write request");
7  is.read("read reply");
```

Claro que esta maneira é demasiado simplista, ignorando aspectos como segurança, fiabilidade das ligações, e tratamento de erros (vulgarmente designados de excepções [javg] no Java). O nosso plano é produzir uma maneira de utilização de Sockets que, mantendo-se o mais parecido possível ao utilizado no Socket Java, permita resolver parte dos problemas enunciados anteriormente (principalmente no que toca a fiabilidade da ligação e ao tratamento de excepções). A razão pela qual optámos por manter a utilização o mais semelhante possível ao Socket Java actual, prende-se pela facilidade com que qualquer programador habilitado para programar com estes Sockets rapidamente se adapta à produção de aplicações que utilizem uma implementação do nosso padrão.

Como pretendemos suportar a comutação de uma conexão para outra, mantendo a interacção entre os dois *hosts* activa, consideramos dois tipos principais de mudança de conexão:

probe Este tipo de mudança ocorre quando a ligação se encontra operacional, mas outro par de endereços com melhores características foi descoberto. Este sistema está periodicamente a monitorizar as ligações em busca de alternativas melhores à conexão (ao par de endereços) actual. Este sistema deve ser chamado de forma explicita, pois apesar de estar a executar em separado, a acção de mudança só é desencadeada, na totalidade, aquando de uma chamada explícita por parte do programador.

excepção Este tipo de mudança ocorre quando a ligação retornou algum erro (excepção), caso em que um novo par de endereços terá que ser escolhido e utilizado. Este sistema é accionado apenas em caso de erro grave e tem de ser também invocado de forma explicita (não é transparente para o programador).

Outro aspecto muito importante é o facto do padrão estar desenhado para funcionar de forma eficaz somente se forem efectuadas várias chamadas de leitura e escrita em sequência, e.g. pedindo um objecto por partes, em vez do todo. Tal foi necessário pois, pedindo um objecto por completo, o primeiro modo de mudança (*probe*) acabaria por não ser possível, já que não existiria modo de chamar o método de mudança de forma explicita, entre leituras-escritas sucessivas. Outro exemplo que funcionaria seria o de fazer a transferência de um conjunto de ficheiros, e.g. todos os ficheiros numa certa directoria, em que se verificava entre pedidos se a ligação continuaria a ser considerada a melhor.

Este detalhe pode ser perceptível através do esquema da Tabela 5.2 e respectiva listagem na Listagem 5.2.

Tabela 5.2: Interacção padrão pedido/resposta

Passo	Descrição
1	Criar Socket, com endereço e porta de destino.
2	Obter canais de input / output do respectivo Socket.
3	Utilizar o canal de output para escrever um pedido para receber todo o objecto.
4	Utilizar o canal de input para ler todos os bytes do objecto.

Listagem 5.2: Interacção padrão Socket Java

```
1  Socket s = new Socket(remote hostname, remote port);
2
3  InputStream is = s.getImputStream();
4  OutputStream os = s.getOutputStream();
5
6  os.write("write request for full object");
7  is.read("read all bytes received at once");
```

Como podemos observar, não se encontra presente nenhum ciclo que permita chamar, de forma explicita, o primeiro método de mudança (*probe*). Portanto, um novo esquema é necessário para resolver esta questão, como o apresentado na Tabela 5.3 e Listagem 5.3.

Tabela 5.3: Interacção padrão pedido/resposta continuado com teste de qualidade

Passo	Descrição
1	Criar *NChannelSocket*, com endereço e porta de destino.
2	Obter canais de input / output do respectivo Socket.
3	Enquanto todo o objecto não tiver sido transferido:
3.1	Utilizar o canal de output para enviar um pedido para receber uma de N partes do objecto.
3.2	Utilizar o canal de input para ler todos os bytes da parte pedida do objecto.
3.3	Verificar explicitamente se o melhor caminho ainda está em uso (modo de mudança *probe*).

Listagem 5.3: Interacção padrão pedido/resposta continuado com teste de qualidade

```
1  Socket s = new Socket(remote hostname, remote port);
2
3  InputStream is = s.getImputStream();
4  OutputStream os = s.getOutputStream();
5
6  do{
7    os.write("write request for 1 of N parts of the object");
8    is.read("read all bytes received for this request");
9
10   if("connection is not the best one")
11     "swith connection";
12 }
13 while("object not transfered");
```

Como discutido anteriormente, ainda falta um detalhe neste esquema, pois este ainda não reflecte o tratamento de falhas. Como pretendemos também conseguir tratar falhas de leitura e mesmo da própria conexão, temos que adicionar algum mecanismo de tratamento de excepções. No Java, este mecanismo está muito presente, sendo que uma

simples implementação utilizando um Socket Java já tem que ser rodeada por *try-catch* [javj]. Este mecanismo permite executar um conjunto de operações sendo que, se algo der errado, consegue apanhar e tratar o erro conforme necessário.

Assim, se adicionarmos ao terceiro esquema (Tabela 5.3) este mecanismo de excepções, ficamos com o esquema presente na Tabela 5.4 e respectiva listagem na Listagem 5.4.

Tabela 5.4: Interacção padrão incluindo tratamento de excepções

Passo	Descrição
1	Iniciar teste ao bloco de código principal:
1.1	Criar Socket, com endereço e porta de destino.
1.2	Obter canais de input / output do respectivo Socket.
1.3	Enquanto todo o objecto não tiver terminado:
1.3.1	Iniciar teste ao bloco de código secundário:
1.3.1.1	Utilizar o canal de output para enviar um pedido para receber uma de N partes do objecto.
1.3.1.2	Utilizar o canal de input para ler todos os bytes da parte pedida do objecto.
1.3.1.3	Verificar explicitamente se o melhor caminho ainda está em uso (modo de mudança *probe*.
1.3.2	Terminar teste ao bloco de código secundário e tratar excepções (modo de mudança *excepção*)
2	Terminar bloco de código principal a testar e tratar excepções.

Listagem 5.4: Interacção padrão incluindo tratamento de excepções

```
1  try{
2    Socket s = new Socket(remote hostname, remote port);
3
4    InputStream is = s.getImputStream();
5    OutputStream os = s.getOutputStream();
6
7    do{
8      try{
9        os.write("write request for 1 of N part of the object");
10       is.read("read all bytes received for this request");
11
12       if("connection is not the best one")
13         "swith connection";
14     }
15     catch(Exception e){
16       "switch connection by exception"
17     }
18   }
19   while("object not transfered");
20 }
21 catch(Exception e){...};
```

Estas excepções que podem ocorrer são principalmente do tipo I/O (input/output), aquando a detecção de uma falha de uma conexão por parte do TCP. As excepções têm que ser tratadas sempre pelo programador, pois dependendo do tipo de aplicação é que as decisões serão tomadas para tratar a excepção. Assim, apenas disponibilizaremos um método de tratamento de excepções (escolhendo outra conexão entre as disponíveis), deixando a cargo de quem programa a aplicação a escolha de como tratar as excepções

que ocorrerem. Na implementação actual, demos também destaque às excepções por *timeout*, quer aquando a tentativa de conectar o Socket, mas também o tempo máximo que se pode esperar na leitura, até qualquer byte ser lido.

O nosso padrão acabou por ter uma estrutura baseada neste último esquema (Tabela 5.4), tendo em consideração qualquer detalhe imposto pela linguagem Java.

5.1 Ferramentas auxiliares

Para ser possível verificar se um determinado canal continua a ser o melhor, este tem que ser periodicamente monitorizado. Como o Java não permite aceder directamente a primitivas no núcleo do sistema, tivemos que encontrar um modo de ter acesso a essas primitivas e, para além disso, a um conjunto de primitivas que permita aceder a determinadas funcionalidades que mesmo o *kernel* tenta esconder. É aqui que entra em funcionamento o módulo de *kernel web10g* e a interface de métodos nativa do Java (JNI) utilizada por nós, que chamamos de ferramentas auxiliares e que passamos a descrever.

5.1.1 *WEB10G*

O *Web10g* [weba] é um *patch* para o *kernel* que permite aceder a um conjunto de informações sobre o funcionamento do TCP. Funciona como uma ponte entre o sistema e as interfaces, monitorizando as conexões em uso e permitindo interceptar um vasto leque de atributos, geralmente escondidos aos utilizadores. A versão utilizada foi a versão 3.0 [webc].

Dos diversos pontos que este *patch* permite aceder, destacamos os que nos são realmente úteis para o correcto funcionamento deste trabalho:

- *ids* dos processos, das conexões e dos utilizadores;
- endereço e porta de origem e de destino (e tipos de endereço);
- número de pacotes recebidos e enviados pelas diversas conexões;
- número de pacotes retransmitidos pelas diversas conexões;
- medições de RTTs (número de RTTs e *Smoothed RTT*) e RTO;
- tempos de medição associados à conexão, i.e. tempo total da conexão e tempos desde a última actividade TCP (envio ou recepção de dados);
- tamanhos correntes das janelas de congestionamento;
- tamanho corrente do MSS.

Este conjunto de variáveis foi encontrada após a análise a dois estudos [PMFR11, HAS02], onde utilizam o *web100* (o *web10g* é o sucessor) e todas as suas funcionalidades para descobrir modos de melhorar a performance das ligações que estudam. Mostram que as conexões podem ser limitadas pela rede, por quem envia e por quem recebe a mensagem, utilizando as variáveis para estudar essas conexões, de modo a puderem efectuar melhoramentos às mesmas.

Após termos definido o tipo de variáveis que pretendíamos, utilizámos a listagem detalhada das variáveis existentes em [MSRH10] de modo a extrair as variáveis que nos poderiam ser úteis. Após esta análise, acabámos por ficar com a lista de variáveis que presentes na Tabela 5.5

Tabela 5.5: Variáveis web10g aplicadas no padrão

Variável	Descrição
cid	ID da conexão em análise.
uid	ID do utilizador que acede às variáveis.
pid	ID do processo que gera os eventos registados.
LocalAddress	Endereço IP origem da conexão.
LocalPort	Porta de origem utilizada.
RemoteAddress	Endereço IP destino da conexão.
RemotePort	Porta de destino utilizada.
SegsOut	Número total de segm1entos enviados.
DataSegsOut	Número de segmentos enviados contendo um tamanho positivo de dados.
DataOctetsOut	Número de octetos enviados contendo um tamanho positivo de dados.
SegsRetrans	Número de segmentos retransmitidos.
OctetsRetrans	Número de octetos retransmitidos.
SegsIn	Número total de segmentos recebidos.
DataSegsIn	Número de segmentos recebidos contendo um tamanho positivo de dados.
DataOctetsIn	Número de octetos contidos nos segmentos recebidos com tamanho positivo de dados, incluindo dados retransmitidos. Nota que não inclui os cabeçalhos do TCP.
ElapsedSecs	Parte em segundos do tempo que passou de actividade da conexão, i.e. mensagens enviadas ou recebidas.
ElapsedMicroSecs	Parte em micro-segundo do tempo passado de actividade da conexão.
CurMSS	Tamanho máximo actual para um segmento (MSS), em octetos.
SmoothedRTT	Valor ponderado do RTT, utilizado no cálculo do RTO.
CountRTT	Número de RTTs lidos.
CurRTO	Valor actual do *retransmit timer* (RTO).
CurCwnd	Tamanho actual da janela de congestionamento, em octetos.

Para além do *patch* propriamente dito, os autores disponibilizam também um conjunto de ficheiros de exemplo, programados em C, que permitem aceder às variáveis colectadas pelo *patch*.

Como sugerido em [bui], primeiro procedemos à instalação e à configuração do *kernel* e do *patch* numa máquina virtual e só depois os instalámos numa máquina física.

Seguimos esta sugestão por questões de precaução, visto que a máquina onde se instalou o *patch* foi a mesma onde o desenvolvimento ocorreu.

Depois de instalar e configurar o *patch* no *kernel*, restou fazer download e instalar a API com as funções e exemplos fornecidas, designada de *userland* [webb].

Após instalada a *userland*, podemos aceder à pasta *[userland]/util/C* onde se encontram os exemplos. Encontramos exemplos de como: ler todas as conexões activas (o *output* mostrado é a medição de todas as variáveis, para todas as conexões); listar as conexões (sem as variáveis, apenas endereços e *ids*); ler apenas uma conexão especifica, dado o seu *ConnID* (*connection id*). Para o desenvolvimento da biblioteca em código nativo (C), aproveitámos o código disponibilizado, principalmente o de aceder a todas as variáveis de todas as conexões, com as devidas alterações necessárias.

O que a chamada de código nativo faz é, dado um *process id*, neste caso da JVM do Java que é o processo pai para o qual nos interessa analisar as conexões criadas, percorremos todas essas conexões, criando uma lista de *ConnStructs* (objecto Java que possui todos

os campos representativos de uma conexão) apenas com as variáveis que nos interessa analisar. Esta lista é enviada para a classe Java que a processa como necessário.

De modo a integrar o código em C gerado por este *patch*, tivemos que utilizar também chamadas a código nativas no Java, sendo a ferramenta escolhida o JNI.

5.1.2 *Java Native Interface*

O JNI [jnib] é um dos modos disponíveis para conseguir aceder a chamadas de métodos nativos. Para ser utilizado, uma classe Java tem que possuir pelo menos um método em que o cabeçalho do mesmo possui a palavra *native* (Listagem 5.5), que é o método que irá ser chamado. O objectivo é integrar o método nativo numa biblioteca que pode ser chamada pela mesma ou outra classe Java, de modo a que as chamadas nativas possam ser processadas.

Listagem 5.5: Exemplo de um método nativo

```
1 /**
2    * Native method
3    * @return error code in case of failure, 0 in case of success
4    */
5   private native int getConnAttributtesFromPID();
```

Apesar de existirem outras opções para comunicação com código nativo (uma delas JNA - *Java Native Access* [jnaa], que permite aceder mais facilmente a uma biblioteca nativa, mas por norma a custo de desempenho [jnab]), decidimos optar pelo uso do JNI. Não só porque tivemos que implementar toda a biblioteca nativa (baseando-nos nos exemplos fornecidos no *userland*, Subsecção 5.1.1), mas porque o desenvolvimento com esta ferramenta é relativamente rápido no caso de já termos uma base de código de exemplo em código nativo (C).

Contudo, este mecanismo (assim como os mecanismos semelhantes) possui alguns problemas, sendo que o JNI tem na gestão de memória (i.e. o vulgarmente chamado de *garbage collector* [javb]) o seu principal problema, já que não faz a gestão de memória das chamadas às bibliotecas nativas (tendo o código nativo, neste caso C, a ter em atenção quando deve libertar os recursos ocupados). Outras questões como erros subtis no uso do JNI, ou a falta de verificação de erros de uma forma correcta, pode tornar instável toda a JVM, podendo inclusive levar ao *crash* da mesma e, por consequência, das aplicações em execução.

As informações acerca do JNI aconselham [jnia] que se envolva o processo de criação da biblioteca nativa num *makefile* [mak] a ser processado sempre que necessário. Neste ficheiro estão presentes as linhas de comando necessárias para uma correcta compilação e integração do código nativo C, de modo a criarmos a biblioteca partilhada para ser chamada através do Java.

Agora que já temos definido o esquema principal em que o padrão se insere, assim como as principais ferramentas que utiliza, vamos apresentar a implementação que fizemos do padrão, começando por apresentar a estrutura do sistema, os métodos e funcionalidades principais do núcleo do *NChannelSocket* e, posteriormente, a aplicação concreta que foi desenvolvida para o testar.

5.2 Estrutura

Um projecto Java de dimensões consideráveis, como é o nosso, costuma seguir uma organização de hierarquia de pastas (denominadas de pacotes). É nestas pastas que os ficheiros de código *.java* são armazenados e organizados. A estrutura criada de pacotes e respectivas classes está representada na Figura 5.1. Todo o núcleo da implementação do padrão encontra-se nos pacotes *nchannelsocket* e *web10gjni*. O primeiro contém todas as classes que implementam o mecanismo de escolha e monitorização dos diversos canais, assim como a classe que representa o Socket por nós desenvolvido (*NChannelSocket.java*). O segundo contém todas as classes necessárias à interacção entre a aplicação Java e o núcleo do sistema operativo (*kernel*), onde se encontra a biblioteca do *web10g*, referenciada na Subsecção 5.1.1. Aqui, encontra-se a classe que tira partido do *Java Native Interface* (Subsecção 5.1.2), assim como o código máquina (*C*), e a biblioteca de código nativa dai resultante (*libweb10gjni_NativeWebInterface.so*). No pacote *implementations* encontram-se as classes que resultaram de uma implementação de teste concreta do padrão desenvolvido (*HTTPDownloader.java* e *FileDownloader.java*). Esta implementação concreta será falada mais em detalhe na Secção 5.5.

Na Figura 5.2, apresentamos o diagrama de classes geral pertencente à parte de núcleo da aplicação (pacotes *nchannelsocket* e *web10gjni*), i.e. sem detalhes dos métodos implementados em cada classe. Na Figura 5.3 está o exemplo do funcionamento do *NChannelSocket* a nível de interacção entre as classes, a aplicação e o sistema.

Em seguida descrevemos um pouco mais em detalhe o papel de cada uma das classes acima listadas:

NChannelSocket.java
: Classe principal do padrão. Tudo o que lide com *Sockets* encontra-se nesta classe. É nesta classe que se encontra a instância do Socket actualmente em uso. Este é retornado sempre que necessitamos comunicar pelo canal estabelecido. Esta classe contém todos os métodos necessários a serem chamados para implementar uma aplicação que tire partido do padrão desenvolvido.

PerformanceManager.java
: Classe fulcral no funcionamento de todo o padrão. Controlada pelo *NChannelSocket*, é nesta classe que se encontram implementados todos os métodos que permitem escolher o caminho inicial a tomar, quando se deve mudar de conexão e que outros caminhos se podem aproveitar. Controla o funcionamento dos dois outros *threads*

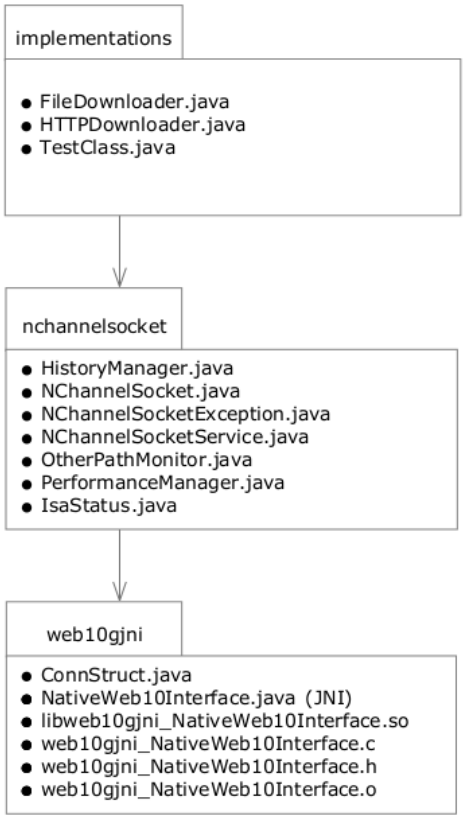

Figura 5.1: Diagrama de pacotes

em funcionamento, representados pelo *NChannelSocketService* e pelo *OtherPathMonitor*, assim como o funcionamento do histórico das conexões em uso/utilizadas (*HistoryManager*).

IsaStatus.java

Classe que permite caracterizar o estado de um determinado endereço, pertencente a um par de endereços. Caracteriza o estado desse endereço segundo uma enumeração [java], sendo que os valores possíveis são: *CONNECTED*, se esse endereço faz parte do par de endereços actualmente em uso; *NOTCONNECTED* caso não esteja a ser utilizado; *BROKE* se esteve envolvido numa falha de uma conexão, quando este estava a ser utilizado; *CHANGED*, se este endereço foi preterido em relação a outro endereço, apesar de estar funcional. Juntamente com o estado de cada endereço está uma marca temporal que indica o momento onde o estado ocorreu. Este tempo é útil para determinar se o endereço mudou de estado dentro de um certo intervalo de tempo, e.g. se falhou à pelo menos X segundos.

HistoryManager.java

Classe responsável por gravar para suporte físico (um ficheiro numa directoria) o histórico dos dados de todas as conexões efectuadas (uma linha por cada par de

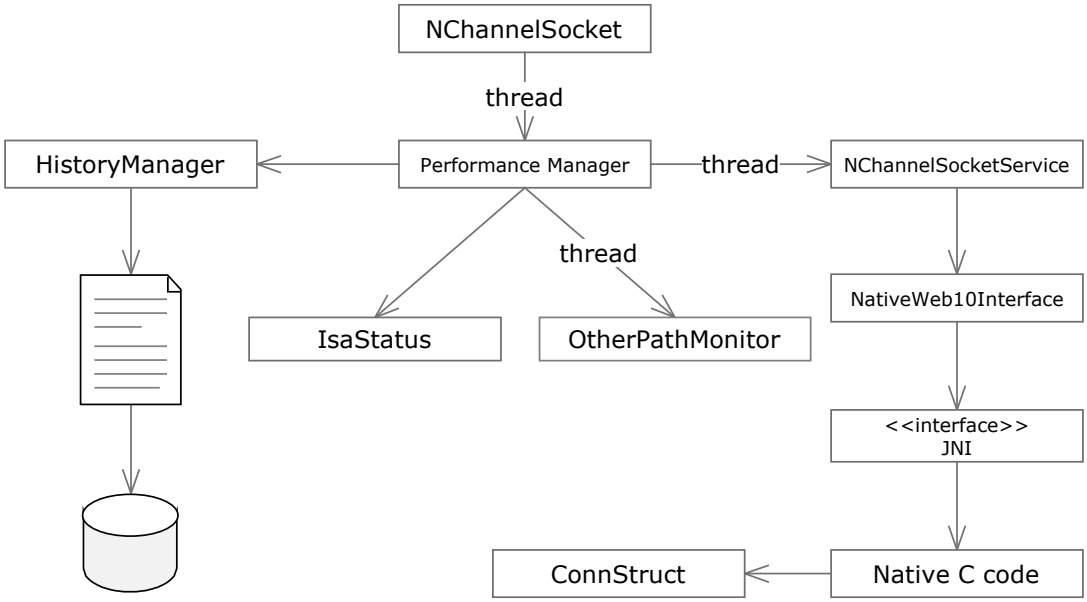

Figura 5.2: Diagrama de classes

endereços, cujas estatísticas serão aglomeradas sempre que for necessário). Responsável também por voltar a carregar deste ficheiro o histórico das ligações no arranque do *NChannelSocket*.

OtherPathMonitor.java

Classe que executa como um *thread*, que permite abrir o que designamos de *probes* (sondas), i.e. apenas se estabelece a ligação, tendo como propósito medir as características do canal (RTT e RTO), sem efectuar transferências de dados. Permite estudar que outras conexões se encontram disponíveis e, caso se verifique, permite-nos avaliar essas conexões, de modo a comutar da conexão actual para uma dessas sondas.

NChannelSocketService.java

Serviço (classe que executa como um *thread*) que permite a monitorização das características de todas as conexões abertas provenientes do processo com identificador igual ao da máquina virtual do Java (JVM), i.e. através do *Web10g*. Controla três estruturas, uma para a última medição das características de todas as conexões abertas, outra para as conexões recentemente medidas (nas últimas dez medições feitas) e uma última com o histórico de todas as conexões efectuadas. A cada 250 milissegundos ocorre uma nova medição. Responsável também por notificar o *PerformanceManager* para actualizar o ficheiro de histórico em disco a cada 20 medições, ou seja, a cada 5 segundos, por questão de *backup*.

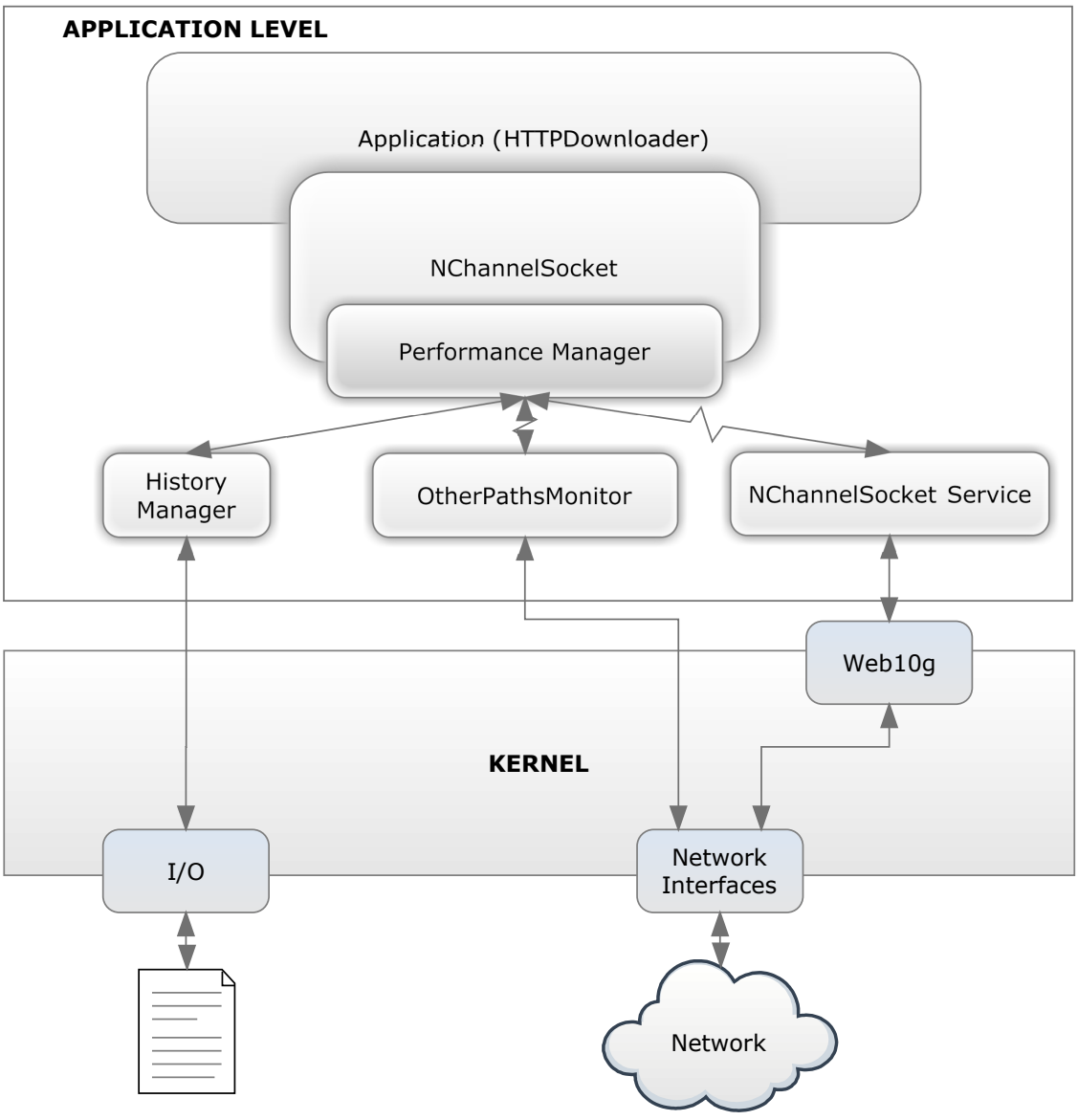

Figura 5.3: Diagrama de arquitectura e execução

NChannelSocketException.java

Excepção que pode ser utilizada, se necessário, juntamente com a aplicação de teste, se tal for útil para diferenciar algum tipo de erro especifico. Neste caso, foi utilizada apenas uma vez na nossa implementação concreta do padrão (Secção 5.5) para um tipo particular de ocorrência. Não é obrigatória de ser utilizada.

NativeWeb10Interface.java

Classe que serve de ponte entre o código Java e as chamadas ao sistema em C, de modo a utilizar a biblioteca nativa *web10g* (Subsecção 5.1.1).

ConnStruct.java

Classe que suporta as diferentes características de uma conexão. É utilizada nas chamadas de código nativo (C), assim como por todo o programa, sempre que é necessário uma estrutura de uma determinada conexão.

5.3 Principais funcionalidades

Em seguida iremos descrever as principais funcionalidades existentes na nossa implementação.

5.3.1 Inicialização do *NChannelSocket*

Para inicializar um Socket Java normal temos vários construtores disponíveis que nos permitem, entre outras coisas, definir endereços de origem e destino a utilizar. Para o nosso caso, resolvemos, por uma questão de extensão de capacidades, permitir chamar quatro construtores diferentes, ou seja, existem 4 modos diferentes de inicializar o nosso *NChannelSocket*. É na inicialização do *NChannelSocket* e conforme o construtor utilizado, que o conjunto dos endereços disponíveis para a escolha do par inicial é calculado. É também nos construtores que é feita a escolha do par inicial a ser utilizado assim como a construção da conexão. Em seguida fica a descrição dos parâmetros de entrada para cada um dos construtores, assim como as principais características, acompanhadas das respectivas listagens de código.

5.3.1.1 Construtor 1 - *Hostname* destino + porta remota

Permite passar como parâmetros um *hostname* e uma porta. Através do *hostname* descobrimos os endereços remotos, anunciados pelo DNS para esse *host* e, juntamente com a porta remota, estabelecemos a conexão. O endereço local é automaticamente escolhido percorrendo todas as interfaces locais existentes, sendo o primeiro disponível utilizado como endereço de origem. A porta de origem é passada com o valor 0, o que indica que uma porta aleatória disponível é escolhida pelo sistema. Este construtor está implementado como mostra a Listagem 5.6

Listagem 5.6: Construtor 1

```
1  public NChannelSocket(String hostname, int port) throws UnknownHostException,
       IOException{
2      this();
3      this.pm = new PerformanceManager();
4
5      InetSocketAddress[] initpair = pm.chooseInitialPairOfAddresses(hostname,
           port);
6
7      while(this.isFirstNotSet()){
8
9        InetSocketAddress localia = initpair[0];
10       InetSocketAddress remoteia = initpair[1];
11       this.socketInUse = new Socket();
12       this.socketInUse.bind(localia);
13
14       try{
15         this.socketInUse.connect(remoteia, CONNECTTIMEOUT);
16         this.socketInUse.setSoTimeout(READTIMEOUT);
17         this.setFirstNotSet(false);
18
19         this.pm.markLocalIsa(localia,IsaStatus.StatusType.CONNECTED);
20         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.CONNECTED);
21         this.pm.updateOtherControllers();
22       }
23       catch(SocketTimeoutException e){
24         dt.printError(e.getMessage());
25         this.pm.markLocalIsa(localia,IsaStatus.StatusType.BROKE);
26         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.BROKE);
27         initpair = pm.chooseInitPair();
28       }
29     }
30   }
```

5.3.1.2 Construtor 2 - Lista de *InetSocketAddress* destino

Se tivermos conhecimento dos endereços de vários servidores onde um objecto esteja replicado (e acessível através da mesma directoria), ou os vários endereços disponíveis para um mesmo servidor, podemos utilizar este construtor. Um *InetSocketAddress* [javd] é um objecto que suporta um *InetAddress* [javc] (representação de um IP em Java) e uma porta. Assim, não temos que fazer pedidos ao DNS pelos endereços de um determinado *host*, utilizando os endereços e portas fornecidos. De notar que é da responsabilidade de quem utiliza o padrão certificar-se que os endereços existem. Caso não consigamos conectar a algum endereço escolhido, o *NChannelSocket* automaticamente tenta escolher outro da lista. Contudo, se nenhum endereço remoto funcionar a transferência termina por excepção. Quanto ao endereço local escolhido, a escolha é feita do mesmo modo que no primeiro construtor (Subsecção 5.3.1.1).

Listagem 5.7: Construtor 2

```
1  public NChannelSocket(Deque<InetSocketAddress> remoteAddresses) throws
       IOException{
2      this();
3      this.pm = new PerformanceManager();
4
5      InetSocketAddress[] initpair = pm.chooseInitialPairOfAddresses(
           remoteAddresses);
6
7
8      while(this.isFirstNotSet()){
9
10       InetSocketAddress localia = initpair[0];
11       InetSocketAddress remoteia = initpair[1];
12
13       this.socketInUse = new Socket();
14       this.socketInUse.bind(localia);
15
16       try{
17         this.socketInUse.connect(remoteia, CONNECTTIMEOUT);
18
19         this.socketInUse.setSoTimeout(READTIMEOUT);
20
21         this.setFirstNotSet(false);
22
23         this.pm.markLocalIsa(localia,IsaStatus.StatusType.CONNECTED);
24         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.CONNECTED);
25
26         this.pm.updateOtherControllers();
27       }
28       catch(SocketTimeoutException e){
29         dt.printError(e.getMessage());
30
31         this.pm.markLocalIsa(localia,IsaStatus.StatusType.BROKE);
32         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.BROKE);
33
34         initpair = pm.chooseInitPair();
35       }
36     }
37
38   }
```

5.3.1.3 Construtor 3 - Endereço origem (*InetAddress*) + porta de origem + booleano de mudança + *hostname* destino + porta remota

Este construtor permite definir um endereço local por onde o utilizador prefere começar a transferência. Assim, utilizando o endereço origem e a porta de origem, juntamente com o mecanismo do primeiro construtor (Subsecção 5.3.1.1) para os endereços remotos,

conseguimos efectuar a conexão. O booleano de mudança é uma variável que, se o seu valor for false, implica que o *NChannelSocket* não irá considerar outros endereços locais existentes para tentativa de conexão. Este modo permite forçar uma interface distinta mas há que ter em atenção que se a interface falhar e esta variável estiver definida para falso, então a transferência termina por excepção, por não ser permitida a escolha de outro endereço origem. Esta escolha fica à responsabilidade do programador.

Listagem 5.8: Construtor 3

```
this();
    this.pm = new PerformanceManager();

    pm.setLocalPreference(localaddress, localport, allowOver);
    InetSocketAddress[] initpair = pm.chooseInitialPairOfAddresses(hostname,
        remoteport);

    while(this.isFirstNotSet()){

      InetSocketAddress localia = initpair[0];
      InetSocketAddress remoteia = initpair[1];

      this.socketInUse = new Socket();
      this.socketInUse.bind(localia);

      try{
        this.socketInUse.connect(remoteia, CONNECTTIMEOUT);

        this.socketInUse.setSoTimeout(READTIMEOUT);

        this.setFirstNotSet(false);

        this.pm.markLocalIsa(localia,IsaStatus.StatusType.CONNECTED);
        this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.CONNECTED);

        this.pm.updateOtherControllers();
      }
      catch(SocketTimeoutException e){
        dt.printError(e.getMessage());

        this.pm.markLocalIsa(localia,IsaStatus.StatusType.BROKE);
        this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.BROKE);

        initpair = pm.chooseInitPair();
      }
    }
```

5.3.1.4 Construtor 4 - Endereço origem (*InetAddress*) + porta de origem + booleano de mudança + Lista de *InetSocketAddress* destino

Este construtor permite definir os endereços locais e a preferência de mudança como no terceiro construtor (Subsecção 5.3.1.3). Quanto ao tratamento dos endereços destino, são agora análogos ao utilizado pelo segundo construtor (Subsecção 5.3.1.2).

Listagem 5.9: Construtor 4

```
1  public NChannelSocket(InetAddress localaddress, int localport, boolean
       allowOver, Deque<InetSocketAddress> remoteAddresses) throws IOException{
2      this();
3      this.pm = new PerformanceManager();
4
5      pm.setLocalPreference(localaddress, localport, allowOver);
6
7      InetSocketAddress[] initpair = pm.chooseInitialPairOfAddresses(
           remoteAddresses);
8
9      while(this.isFirstNotSet()){
10
11       InetSocketAddress localia = initpair[0];
12       InetSocketAddress remoteia = initpair[1];
13
14       this.socketInUse = new Socket();
15       this.socketInUse.bind(localia);
16
17       try{
18         this.socketInUse.connect(remoteia, CONNECTTIMEOUT);
19
20         this.socketInUse.setSoTimeout(READTIMEOUT);
21
22         this.setFirstNotSet(false);
23
24         this.pm.markLocalIsa(localia,IsaStatus.StatusType.CONNECTED);
25         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.CONNECTED);
26
27         this.pm.updateOtherControllers();
28       }
29       catch(SocketTimeoutException e){
30         dt.printError(e.getMessage());
31
32         this.pm.markLocalIsa(localia,IsaStatus.StatusType.BROKE);
33         this.pm.markRemoteIsa(remoteia,IsaStatus.StatusType.BROKE);
34
35         initpair = pm.chooseInitPair();
36       }
37     }
38
39   }
```

5.3.2 Controle de histórico de conexões

Na nossa implementação encontra-se um histórico de todas as conexões efectuadas (uma linha por par de endereços). Este histórico está gravado em suporte físico (disco) e é utilizado, quando existe, para melhorar a escolha inicial de um par de endereços. A leitura e a escrita estão a cargo da classe *HistoryManager.java*, que é controlada pelo *PerformanceManager.java* (e, por sua vez, pelo *NChannelSocket.java*). A escrita do histórico actualizado é feita de 5 em 5 segundos, por questões de *backup*, e no final da aplicação, quando a transferência termina. Como referido, é guardado uma linha para cada par de endereços utilizados, linha essa que contém em formato texto a estrutura da respectiva conexão (do objecto Java que a representava).

No arranque do *NChannelSocket*, este ficheiro de histórico é lido e é feita a reconstrução de cada uma das conexões gravadas para os respectivos objectos Java que caracterizam uma conexão (*ConnStruct*). Após a leitura do ficheiro de histórico para memória ficamos com uma estrutura que é uma lista de *ConnStructs*.

5.3.3 Escolha inicial de um par de endereços

As listas de todos os endereços (*InetSocketAddress*) locais e remotos, definidos na instanciação do *NChannelSocket* (Subsecção 5.3.1), constituem as fundações para a escolha de qualquer par de endereço durante toda a execução da aplicação. Ou seja, não há novos pares a entrarem ou pares a saírem durante a vida do objecto. Apenas poderão ficar pares temporariamente indisponíveis para uso em caso de, por exemplo, falha. Assim, depois de definidos quais os endereços possíveis para efectuar a ligação podemos proceder à escolha do que será o par inicial para arrancar a transferência.

Se existir histórico (Subsecção 5.3.2), então podemos tentar acelerar o processo de escolha inicial, percorrendo todos pares de endereço que combinem com a lista de pares de endereços origem e de endereços destino. Dos pares de endereços presentes no histórico que cumpram os requisitos iremos escolher o que para nós considerarmos como sendo o melhor desses pares. Para tal, aplicamos um conjunto de regras, através da medição das diversas características presentes numa determinada conexão (que surgem do uso do *web10g* - Subsecção 5.1.1), e que passamos a anunciar.

Dado um conjunto de conexões presentes no histórico, já devidamente carregadas para memória, $C = c_1, c_2, ..., c_x$, em que cada elemento possui todas as variáveis de uma conexão, denominamos as diversas características por $c_x.nome$, em que $nome$ corresponde ao nome das variáveis utilizadas, presentes na Tabela 5.5.

Dadas uma lista dos endereços origem $O = o_1, o_2, ..., o_n$ e uma lista dos endereços destino $D = d_1, d_2, ..., d_z$ disponíveis.

Para todas as conexões em que $c_x.LocalAddress = o_n$ e $c_x.RemoteAddress = d_z$, escolhemos como melhor a conexão que tenha um melhor *throughput* ou que tenha um *SmoothedRTT* mais baixo, ou em que o *SenderPacketLoss* (Subsecção 5.3.6) seja mais baixo ou que o RTO seja mais baixo, e onde o estado da conexão quando terminou não seja por

erro (*BROKE*), ou se for, que o tenha sido há mais tempo do que o mínimo exigido, que no caso da nossa implementação são 30 segundos.

Caso não exista histórico, apenas percorremos as listas de endereços origem e destino e retornamos o primeiro par disponível.

5.3.4 Escolha posterior de outro par de endereços

A escolha posterior de outros pares de endereços, irá ocorrer se:

- o par de endereços actualmente em uso levantar excepção, e.g. *timeout* de leitura ou de conexão;
- for detectado através de *probing* que, apesar do par actual de endereços estar funcional, existe uma alternativa potencialmente melhor que deve ser testada.

No caso de excepção, o par de endereços é marcado como quebrado (*BROKE*), não sendo sequer utilizado para futuras tentativas de conexão, pelo menos durante 30 segundos e somente se mais nenhum par de endereços estiver disponível. No caso em que existe uma melhor alternativa, o par de endereços é marcado como estando modificado (*CHANGED*), sendo que neste caso se aplica a mesma regra de esperar 30 segundos por nova tentativa de conexão.

Para escolher outro par, caso tenha sido por excepção, as regras de escolha de outro endereço são semelhantes às utilizadas para escolher um endereço inicial, em que será ignorado o par de endereços que acabou de falhar, e será escolhido outro par segundo as regras aplicadas anteriormente. Aqui, o "histórico"de conexões a percorrer já estará mais actualizado com o aglomerar das conexões actualmente em uso.

Caso a escolha seja feita por existir uma melhor alternativa, iremos percorrer as conexões a funcionar actualmente (a que está a ser utilizada para transferência e as sondas (*probes*)), e comparamos essas conexões, escolhendo a que possua um menor *SmoothedRTT* ou um menor *CurRTO*. Neste caso, apenas utilizamos estas duas variáveis porque as sondas apenas estabelecem a conexão, não fazendo de facto transferências de dados, pelo que não é fiável comparar *throughputs* ou *packetLoss*. Já o *SmoothedRTT* e o *CurRTO* dão uma indicação da latência da conexão, mesmo não estando a transferir dados. Claro que não é 100% fiável e garantido, mas permite comutar para uma conexão que em principio estará em melhores condições.

5.3.5 Probing

O *probing* (sondagem) foi um mecanismo que adicionámos que permite abrir conexões, sem transferir dados, num *thread* a executar em paralelo e que permite sondar todos os outros pares de endereços disponíveis, e que não estejam actualmente em uso. A criação destes *probes* está sempre dependente da escolha de um novo par de endereços para transferência. Assim, quando um par inicial de endereços é escolhido, ou sempre que um par posterior de endereços for escolhido, este *thread* é actualizado.

O seu funcionamento é relativamente simples. Sempre que se justifica é passado para este *thread* a lista de todos os endereços locais e remotos que não estão conectados (*NOT-CONNECTED*) nem *BROKE* (ou se estiverem, só são considerados ao fim dos tais 30 segundos). A função do *thread* é criar conexões normais (criando Sockets Java comuns) e mantendo as conexões abertas, de modo a que o sistema de monitorização de conexões (*NChannelSocketService*) consiga monitorizar os atributos correspondentes a estas sondas. Estas conexões mantém-se abertas para darmos tempo ao sistema de ler os atributos da conexão. Podíamos ter seguido uma abordagem um pouco diferente, e manter abertas as sondas apenas pelo tempo necessário a uma leitura. Como o mecanismo de sondas foi das últimas coisas a ser implementadas, esta aspecto ficou como algo a melhorar no futuro (Capítulo 7).

Assim como na criação, também o fecho das sondas está relacionado com problemas ou mudanças na conexão activa. Sempre que existe o fecho da ligação activa (ou porque levantou excepção, ou porque outra possivelmente melhor foi detectada), todas as sondas são fechadas.

Este mecanismo foi principalmente criado com o intuito de, mesmo estando uma conexão activa a funcionar correctamente, esta pode não ser a melhor de todas as disponíveis. Se apenas nos focássemos nos mecanismos de mudança de conexão (Subsecção 5.3.6) quando a conexão piora ou tem algum problema, poderíamos nunca atingir a melhor das conexões disponíveis. Este mecanismo, apesar de poder introduzir algum *overhead* ao sistema, acaba por compensar, como veremos no capítulo seguinte.

5.3.6 Decisão de mudança

Existem diversas condições de teste à mudança da conexão em uso. Esta mudança ocorre sempre que a conexão está funcional, mas pretendemos testar se entretanto piorou (e.g. diminuiu o *throughput* ou aumentou a latência) ou se existe um caminho alternativo considerado melhor.

O método que permite testar se é aconselhável esta mudança tem o nome de *switchConnection()*. Neste método são chamados vários métodos auxiliares que constituem no seu todo as diferentes condições de teste. Esta função retorna um número que está associado a um tipo de mudança: 0, sem mudança; 1, problemas em quem envia; 2, problemas em quem recebe; 3, mudança aleatória. Em seguida apresentamos uma descrição do que consiste cada um dos métodos auxiliares.

grownUpConnection()

Uma conexão é considerada "madura"se o número de RTTs transmitidos for maior que *NUMRTT* (actualmente 100) ou o tempo passado da conexão for igual ou superior a *SECONDSTOGROW* (5 segundos). Se este caso se verificar, a função retorna verdadeiro (*true*).

availableAlternatives()

Se existir um endereço (local ou remoto) que esteja desconectado (estado *NOTCONNECTED*) ou que está modificado ou quebrado (*CHANGED* e *BROKE*, respectivamente) mas há mais que 30 segundos (definido no sistema), retorna verdadeiro (*true*).

senderDependent()

Se $\frac{octetosRecebidos}{octetosEnviados} \leq 1$ indica que enviei mais do que recebi logo a conexão, do meu ponto de vista, está dependente de eu enviar, logo designo que é *sender dependent* (dependente de quem envia).

isWindowInRecoveryState()

Se o tamanho da janela de congestionamento $\leq 2 *$ tamanho de 1 MSS retorna verdadeiro (*true*).

highSenderPacketLoss()

Se $recentSenderPacketLoss > \alpha * pastSenderPacketLoss$ retorna verdadeiro. *recentSenderPacketLoss* é caracterizado pelo valor da taxa de perca de pacotes da conexão ($\frac{segmentosTransmitidos}{segmentosRetransmitidos}$) das últimas dez medições, enquanto o *pastSenderPacketLoss* é caracterizado pela taxa de perca de pacotes desde que temos registo desta conexão. O valor de $alpha$ está definido como 0.2, ou seja, se o valor recente da taxa de perca de pacotes for 20% superior ao valor passado, retorna verdadeiro.

receiverDependent()

Se $\frac{octetosRecebidos}{octetosEnviados} > 1$ indica que recebi mais do que enviei, logo do meu ponto de vista eu sou mais receptor que emissor, logo a conexão é *receiver dependent* (dependente de quem recebe).

receiverThroughputDecayed()

Se $recentReceiverThroughput < \alpha * pastReceiverThroughput$ retorna verdadeiro. O valor de *recentReceiverThroughput* é calculado pelo *throughput* ($\frac{octetosEnviados}{tempoDaConexao}$) das ultimas dez medições, enquanto que o *pastReceiverThroughput* é o *throughput* desde que há registo da conexão. O valor de $alpha$ está definido como 0.2, ou seja, se o valor recente do *throughtput* for 20% inferior ao valor passado, retorna verdadeiro.

senderThroughputDecayed()

Se $recentSenderThroughput < \alpha * pastSenderThroughput$ retorna verdadeiro. O valor de *recentSenderThroughput* é calculado pelo *throughput* ($\frac{octetosRecebidos}{tempoDaConexao}$) das últimas dez medições, enquanto que o *pastSenderThroughput* é o valor desde que a conexão está registada. O valor de $alpha$ está definido como 0.2, ou seja, se o valor recente do *throughtput* for 20% inferior ao valor passado, retorna verdadeiro.

Mudança aleatória

Neste caso, temos um número aleatório que é como um "medidor de risco aleatório", ou seja, determina com mais ou menos probabilidade se devo arriscar a mudança para outra conexão. Se verificamos que existe uma conexão melhor que a conexão actual (utilizando as sondas), aumentamos a probabilidade de mudança para 60%. Caso tal não se verifique, apenas arriscaremos uma mudança para outra

ligação com 10% de probabilidade.

Estas funções trabalham em conjunto para chegarmos às condições de teste de mudança, que passamos a anunciar:

1. se a conexão não é madura (!*grownUpConnection*) não aconselha mudar (retorna o valor 0);
2. se não há alternativas (!*availableAlternatives*) não aconselha mudar (retorna o valor 0);
3. se a conexão for *senderDependent* e a janela estiver ainda em recuperação (*isWindowInRecoveryState*);
   (a) se quem envia registou um grande aumento no *packet loss* (*highSenderPacketLoss*), aconselha mudar (retorna o valor 1);
   (b) se o *throughput* do ponto de vista de quem envia piorou significativamente, *receiverThroughputDecayed* (quem envia interessa saber o ritmo a que se consegue enviar dados ou seja, o ritmo a que o receptor os recebe), aconselha mudar (retorna o valor 1);
4. se a conexão for *receiverDependent* e o *throughput* de quem envia (quem recebe interessa-lhe analisar o ritmo de quem envia, ou seja, o ritmo a que recebe dados) piorou significativamente (*senderThroughputDecayed*), aconselha mudar (retorna o valor 2);
5. se passar por todos os pontos acima, é porque a conexão pode ser mudada, mas está a funcionar correctamente, sendo que neste caso é aplicada a mudança aleatória descrita anteriormente (retorna o valor 3, caso a mudança aleatória se verifique).

Na Listagem 5.10 encontra-se o código da implementação actual do método de escolha, com todas as situações acima explicadas.

Listagem 5.10: Decisão de mudança

```
1  private int switchConnection(ConnStruct recentConn, ConnStruct pastConn){
2
3      /*
4       * If this connection is not grown up, return false.
5       */
6      if(!this.grownUpConnection(recentConn)){
7        this.setLastSwitchType(0);
8        return 0;
9      }
10
11     /*
12      * If there are no more alternatives to this connection, return false.
13      */
14     if(!this.availableAlternatives(recentConn)){
15       this.setLastSwitchType(0);
16       return 0;
17     }
```

```
18
19      /*
20       * If this connection is sender dependent and is in a recovery state
21       * then we test two other conditions:
22       * - if the sender packet loss has recently risen, return 1;
23       * - because this is the sender, it is interesting to check if the
             throughput
24       * of the receiver has recently decayed, meaning that the throughput in
25       * which the sender is sending packets has also decayed. Returns 1;
26       *
27       * If this connection is receiver dependent and if the rate at which I am
28       * receiving packets has decayed, this probably means that the rate in
             which
29       * the sender is sending packets has also decreased, and we should try
30       * another connection. Returns 2;
31       */
32      if(this.senderDependent(recentConn) && this.isWindowInRecoveryState(
             recentConn) ){
33
34        if(this.highSenderPacketLoss(recentConn,pastConn) || this.
               receiverThroughputDecayed(recentConn,pastConn)){
35          this.setLastSwitchType(1);
36          return 1;
37        }
38      }
39      else if(this.receiverDependent(recentConn) && this.senderThroughputDecayed(
             recentConn,pastConn)){
40        this.setLastSwitchType(2);
41        return 2;
42      }
43
44      /*
45       * If we got this far, then the connection is working properly.
46       * We can try and test if there are better connections using probes.
47       * This measurements only uses the SmoothedRTT and CurRTO.
48       * With a small percentage of times, we compare the probes to the current
49       * connection, and if a better one has been detected, we switch
50       * connections.
51       */
52      double random = Math.random();
53
54      boolean isABetterOne = isThereABetterPath(recentConn);
55
56      double chosenProb;
57
58      if(isABetterOne)
59        chosenProb = HIGHPROB;
60      else
61        chosenProb = SMALLPROB;
62
```

```
63      if(random < chosenProb){
64        this.setLastSwitchType(3);
65        return 3;
66      }
67
68      this.setLastSwitchType(0);
69      return 0;
70    }
```

5.4 Implementação do padrão

Como já foi referido no capítulo anterior, o núcleo do padrão encontra-se totalmente implementado em Java, estruturado num esquema de classes como apresentado na Secção 5.2. Para podermos verificar o seu funcionamento, implementámos o padrão numa aplicação que utiliza pedidos HTTP 1.1 [RUJ+99] para fazer o download de um objecto.

Como tem vindo a ser explicado, o nosso padrão segue o esquema apresentado na Tabela 5.4. Assim sendo, para implementar uma aplicação de teste funcional, necessitamos esclarecer alguns pontos:

- optámos por implementar uma aplicação que faça pedidos parciais HTTP 1.1 (pedido com código 206) a um servidor, por permitir fazer vários pedidos sobre um mesmo objecto, sendo útil para testar os mecanismos desenvolvidos;
- optámos por utilizar HTTP (mais especificamente, pedidos cliente-servidor), devido ao tráfego actualmente na Internet ser maioritariamente deste tipo [GPSH12], mas qualquer outro protocolo é utilizável, desde que a aplicação se adeqúe ao padrão por nós desenvolvido;
- dos muitos campos no cabeçalho de um pedido HTTP [htt], importa principalmente o campo *Range* para pedir a parte do objecto que nos interessa processar e o campo *Keep-alive* no caso de pretendermos uma ligação mais longa;
- irão ser relevantes os cabeçalhos de resposta *Connection*, *Content-length* e *Content-range*, de modo a sabermos o estado da conexão com o servidor, o tamanho do corpo da mensagem e o range do número de bytes pedidos e tamanho total do objecto, respectivamente;
- não utilizamos quaisquer bibliotecas Java de tratamento/*parsing* de pedidos HTTP, mantendo-nos somente com a utilização do fornecido pelos Sockets, sendo que os pedidos GET são feitos à mão e enviados pelo canal de saída do Socket como um *array* de bytes;
- o mesmo se aplica aos tratamentos das respostas, recebemos e tratamos um *array* de bytes.

Em seguida, mostramos a estrutura de uma aplicação semelhante ao por nós implementada (retirar um objecto por partes), que implemente os Sockets pré-definidos no

Java (i.e. Java Sockets), Listagem 5.11, versus uma aplicação que implemente os Sockets por nós definidos (*NChannelSocket*), Listagem 5.12. Apesar de ambos os exemplos se basearem no nosso padrão, na Listagem 5.11 não é possível utilizar os mecanismos de verificação ou comutação de melhores caminhos, visto não existirem. É essa a principal diferença que nos destinge entre o existente no Java e a nossa implementação.

Listagem 5.11: Implementação do Java Socket

```
1
2  Socket socket = null; // Reference has to be outside of try-catch, otherwise "
       finally" wouldn't see it.
3
4  try{
5    socket = new Socket("remote_hostname", remote_port);
6
7    do{
8      try{
9        // ...
10       // Make necessary read and write operations, like sending some request or
             getting some response
11       sendRequest(socket);
12       // ...
13       getResponse(socket);
14       // ...
15     }
16     catch( Exception e ){
17       // ...
18     }
19   while( some condition );
20
21 catch( Exception e ){
22   // ...
23 }
24 finally{
25   // Close connection
26   socket.close();
27 }
```

Listagem 5.12: Implementação do Java NChannelSocket

```
1
2  NChannelSocket ncs = null;
3
4  try{
5
6    ncs = new NChannelSocket("remote_hostname", remote_port);
7
8    do{
9      try{
10
11       // ...
```

```
12        // Make necessary read and write operations, like sending some request or
              getting some response
13        sendRequest(ncs.getSocketInUse());
14        // ...
15        getResponse(ncs.getSocketInUse());
16        // ...
17
18        if( Best connection is not being used ){
19          ncs.switchSocket();
20        }
21      }
22      catch(Exception e){
23        // If socket throws an exception, we must treat it.
24        ncs.switchSocketEx(); // Special type of Socket switching.
25        // ...
26      }
27    }
28    while( some condition );
29  }
30  catch( Exception e ){
31    // ...
32  }
33  finally{
34    // Closes current socket being used, and terminates all services.
35    ncs.close();
36  }
```

Como podemos observar, existem muitas semelhanças na utilização do *NChannelSocket*, o modo de ler e escrever no Socket é semelhante, utilizando em vez do objecto Socket completo, o objecto Socket que internamente ao *NChannelSocket* está a ser utilizado. É esta estrutura que definimos como sendo o nosso padrão de utilização do *NChannelSocket*. Criamos o Socket, e enquanto todas as transferências ou todos os bocados de uma mesma transferência não terminarem, envio o próximo pedido, recebo e trato a resposta, e testo os canais em busca de uma melhor conexão à que actualmente está em uso. No caso de existirem tratamentos de excepção (o que é quase obrigatório para o bom funcionamento de uma aplicação deste tipo), disponibilizamos um método que permite tentar encontrar outra alternativa àquela que acabou de falhar (outro par de endereços).

Os métodos presentes de *sendRequest* e *getResponse* são um exemplo de métodos que iriam enviar um pedido ou esperar e tratar uma resposta. De notar que se trata apenas de um exemplo e como estes métodos são na realidade implementados dependerá sempre da aplicação na qual o padrão está a ser aplicado.

5.5 Implementação concreta do padrão

Em seguida, descrevemos a implementação actual do padrão, inserido numa aplicação de teste. Esta aplicação permite fazer o *download* de um ficheiro de um determinado servidor. O código aqui presente teve como base o esquema do código utilizado na Listagem 5.12.

```
1
2     String urls = "http://10.171.64.12/thesis.pdf";
3
4     // The local filename
5     String filename = "thesis.pdf";
6
7     NChannelSocket ncs = null;
8
9     try {
10      // Instance of the file downloader helper class
11      FileDownloader fd = new FileDownloader(urls,filename,32768);
12      fd.isDebug = false; // Prints or not aditional debug information
13
14      // Test with virtual machines
15      Deque<InetSocketAddress> deq = new LinkedList<InetSocketAddress>();
16      // Add all known addresses to a queue
17      deq.addLast(new InetSocketAddress(InetAddress.getByName("10.171.64.10")
            ,80));
18      deq.addLast(new InetSocketAddress(InetAddress.getByName("10.171.64.11")
            ,80));
19      deq.addLast(new InetSocketAddress(InetAddress.getByName("10.171.64.12")
            ,80));
20      // Initialize NChannelSocket with the given local interface and remote
            addresses
21      ncs = new NChannelSocket(InetAddress.getByName("10.170.133.125"),0,false,
            deq);
22
23      do{
24        try{
25          fd.sendRequest(ncs.getSocketInUse());
26          fd.getResponse(ncs.getSocketInUse());
27          // Test connection
28          if(!ncs.bestSocketBeingUsed()){
29            ncs.switchSocket();
30          }
31        }
32        catch(NChannelSocketException e){
33          dt.printError(e.getMessage());
34          ncs.switchSocket();
35        }
36        catch(IOException e){
37          dt.printError(e.getMessage());
38          ncs.switchSocketEx();
```

```
39          }
40
41        }while(fd.isNotFinished());
42
43        fd.writeResultToFile();
44
45      } catch (UnknownHostException e) {
46        e.printStackTrace();
47      } catch (IOException e) {
48        e.printStackTrace();
49      } catch (IllegalArgumentException e) {
50        e.printStackTrace();
51      } catch (URISyntaxException e) {
52        e.printStackTrace();
53      }
54      finally{
55        try {
56          if(ncs != null)
57            ncs.closeFinished();
58
59        } catch (IOException e) {
60          dt.printError("error class: "+e.getClass());
61          dt.printError("force ending...");
62
63        }
64      }
```

5.6 Sumário

Neste capítulo apresentámos a implementação concreta do padrão desenvolvido e do respectivo *framework*. Partimos dum esquema que utiliza o Java Socket comum (Esquema 5.1) chegando a um esquema que representa algo mais semelhante ao utilizado pelo padrão implementado (Esquema 5.4).

Em seguida, foram explicadas duas das ferramentas auxiliares que utilizamos, o *Web10g* (Subsecção 5.1.1) e o JNI (Subsecção 5.1.2), sendo que sem estas duas utilidades o correcto funcionamento do *framework* não seria de todo possível. Posteriormente é explicada a estrutura de todo o *framework*, as classes mais relevantes e o seu funcionamento (Secção 5.2). Na secção seguinte (Secção 5.3) são explicadas as principais funcionalidades criadas e implementadas, desde a inicialização do mecanismo, escolha inicial e posterior de canais, decisões de mudança, sondas e mudança aleatória.

Por fim, mostramos uma possível implementação do Java Socket (Secção 5.4) e a implementação concreta do padrão (Secção 5.5).

6

Avaliação e Resultados

Neste capítulo explicamos os ambientes de teste que foram montados para avaliar o nosso trabalho, os cenários de teste que foram executados e, por fim, os resultados obtidos e sua explicação. Na explicação dos resultados explicamos também o porquê da ocorrência de certos casos, casos estes verificados ao longo dos testes e que por vezes implicaram mesmo a sua repetição.

6.1 Ambientes de teste e configuração

Os testes que serão explicados na Secção 6.2 envolvem a transferência de dois ficheiros, com tamanhos de 19.8MB e 139.4MB, respectivamente (que iremos referir como o ficheiro de 20MB e de 140MB, para simplificar a escrita e compreensão). Foram ainda criados dois ambientes de teste, onde o que difere é o sitio remoto onde o ficheiro pretendido está alojado, i.e. varia os endereços que estão envolvidos e o número de endereços disponíveis.

Para configurar o primeiro ambiente de teste necessitamos também de configurar um sistema denominado *Dummynet* [Riz]. Este sistema permite emular ligações entre dois extremos, ignorando a complexidade do grafo total da Internet. Segundo [CR10], o *Dummynet* suporta *multipath*, simulação de perda de pacotes e falhas de encaminhamento. Pode-se emular uma pequena rede dentro de um mesmo sistema, com caminhos com vários saltos (*hops* entre *routers*), ou então sistemas mais complexos entre várias máquinas. Tem como ponto forte não necessitar do modelo do grafo complexo da rede real e, como ponto fraco, a menor veracidade dos dados, em relação ao obtido num simulador (deixa um pouco a dinâmica da Internet de parte). No Linux está inserido no módulo *ipfw* [ubuc], e foi utilizado na máquina de origem (e também instalado nas máquinas virtuais, apesar de não vir a ser necessário) para permitir configurar três canais que de outra

forma não seria possível configurar, por ser complexa a montagem de uma infraestrutura de teste como a desejada.

Assim, o computador que representa o ponto de origem da conexão é o mesmo que foi utilizado no desenvolvimento de todo o projecto. As suas características estão listadas abaixo:

- processador: Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz;
- memória: 6GB DDR3;
- sistema operativo: Linux, distribuição de *Ubuntu*, versão 11.10 (*kernel* 3.0.16), com as devidas alterações e *patches*, i.e. web10g (Subsecção 5.1.1) e *Dummynet/ipfw*;
- interface Ethernet 1GBits e interface wireless com o standard *IEEE 802.11bgn* [ubub].

O primeiro ambiente de teste consiste em uma máquina de origem, com um endereço origem (para este teste apenas usa a Interface Ethernet) e três maquinas virtuais de destino com IP distintos, máquinas estas que são servidores Apache [apa] em funcionamento. Estas máquinas são conectadas através da rede interna do departamento, e acessíveis somente por cabo (devido à configuração da rede física), como observado na Figura 6.1. Como existe uma rede entre o computador de origem e estas máquinas (a rede interna do departamento), utilizamos o *Dummynet* para configurar os canais, permitindo-nos controlar as características das diversas ligações. A ligação real que se pretende simular é representada pela Figura 6.2. Os campos que utilizamos para configurar cada uma das ligações através do *Dummynet* são: a capacidade do canal; o rácio de perda de pacotes; o RTT; e o tamanho da fila de espera (Tabela 6.1). Quando testámos o *Dummynet* reparámos que modificar demasiado o valor da fila de espera tornava as ligações instáveis, sendo que este valor se manteve constante, assim como o valor da taxa de perca de pacotes.

Tabela 6.1: Características dos canais usando *Dummynet*

Nome	Descrição	Capacidade (Kbps)	Atraso (RTT/2) (ms)	Tamanho da fila (entradas)	Taxa de perca de pacotes
canal-10	canal pior	700	30	15	0.01
canal-11	canal intermédio	900	30	15	0.01
canal-12	canal melhor	1000	10	15	0.01

A definição que utilizámos para descrever a ligação podia ter passado por caminho, *link* ou canal. Todos têm uma semântica particular. Um caminho pode ser um caminho na Internet, passando por vários intermediários. Um *link* e um canal podem ser interpretados de igual modo, mas como o nome do *framework* desenvolvido foi *NChannelSocket*, decidimos por bem continuar na mesma terminologia. Assim, todas as ligações terão a descrição de canal.

Esta configuração partiu do princípio de pretendermos uma ligação considerada pior,

uma intermédia e uma melhor. Como podemos reparar a capacidade da ligação intermédia é muito próxima da melhor ligação. Isto deve-se a querermos conseguir testar se o *NChannelSocket* tiver que escolher uma delas, se mesmo com uma pequena diferença escolheria a melhor. O equivalente se passa entre a pior e a intermédia, que possuem o mesmo atraso, 30ms. Ao escolher uma que não seja a melhor, queremos saber se o sistema escolhe correctamente a que possui maior capacidade, já que o RTT é o mesmo nas duas.

Neste primeiro ambiente de teste, cada uma das máquinas tem o mesmo ficheiro, o de 20MB, acessível externamente pela mesma directoria. Esta configuração permite representar um servidor replicado, com vários endereços disponíveis.

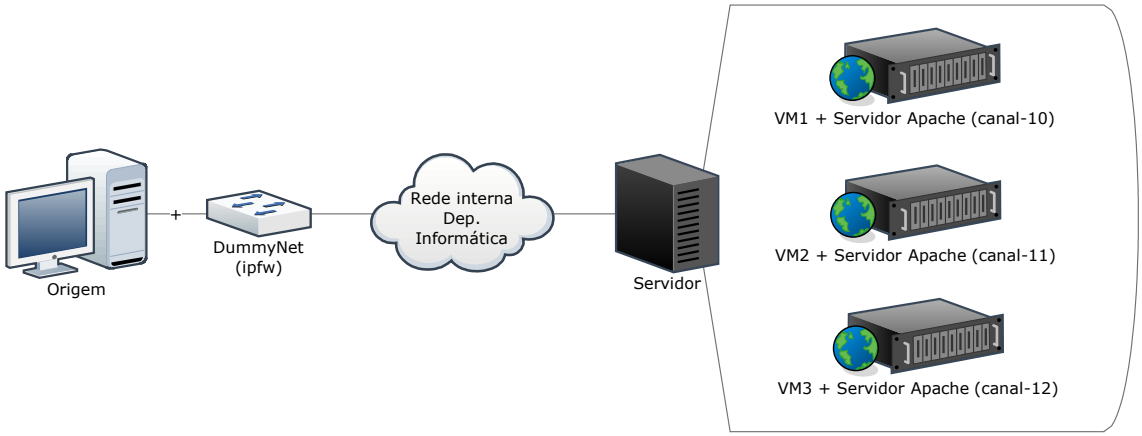

Figura 6.1: Configuração da rede no ambiente de teste 1

O segundo ambiente de teste necessário para validar o funcionamento do sistema desenvolvido, está relacionado com o teste de várias interfaces, agora de origem. Neste segundo ambiente o ficheiro utilizado foi o ficheiro com 140MB de tamanho e foi utilizado um servidor externo à rede do departamento/instituição. Do servidor apenas sabemos que está em Espanha e possui um sistema Unix instalado com um servidor Apache. Este segundo ambiente está representado na Figura 6.3. O computador de origem está conectado a um *router*, quer por Ethernet, quer por Wireless, fazendo parte da mesma rede (apenas possui um endereço IP diferente para cada interface). A rede utilizada tem uma velocidade máxima contratada de 30MBit/s.

Para efeito de teste da nossa implementação (Secção 5.5) criámos uma classe de teste (*TestClass.java*, Figura 5.1) que corre a aplicação de teste as vezes necessárias, gravando os resultados em ficheiros. Estes ficheiros guardam todos os eventos relevantes ao longo da conexão, sempre acompanhado do momento em que o evento ocorreu, na forma de *timestamps* (marcas temporais). Permite-nos medir, de forma precisa, as diversas ocorrências como o início e término da transferência, os momentos de mudança de conexão e detecção de erros ou ainda os momentos de recuperação de falhas. Cada marca temporal

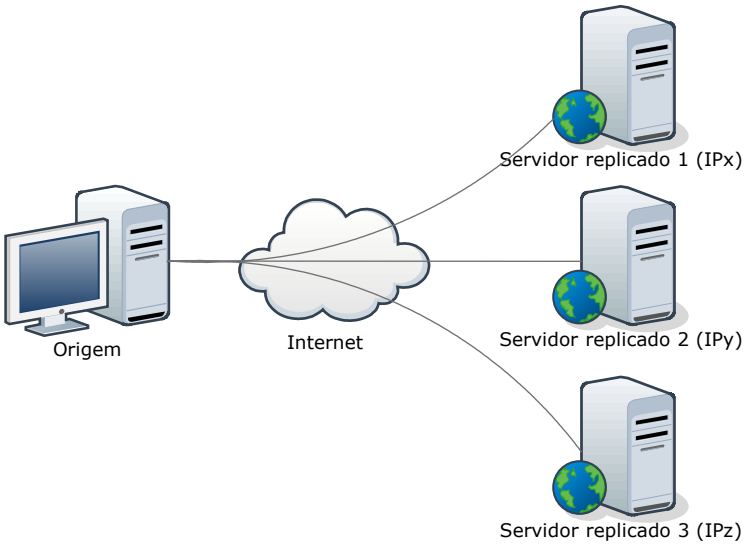

Figura 6.2: Modelo do ambiente de teste 1

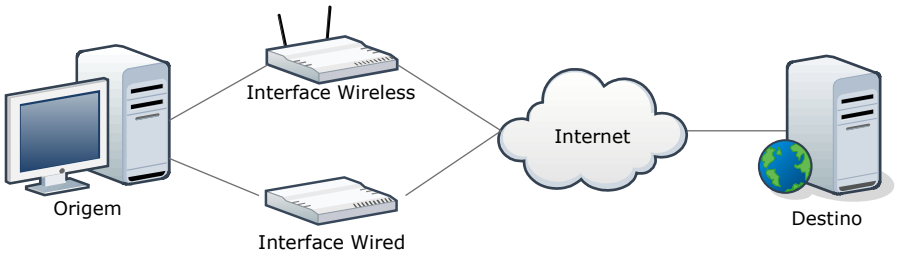

Figura 6.3: Configuração da rede no ambiente de teste 2

é sempre criada utilizando o próprio relógio do sistema, e cada marca temporal encontra-se em milissegundos (representa o tempo em milissegundos desde 1-1-1970). Sempre que numa das tabelas abaixo surja o tempo em milissegundos convertido em segundos, essa conversão é feita dividindo o tempo em milissegundos por 1000. O tempo não é apresentado num formato horário normal *hh:mm:ss*, mas continua perceptível, sendo que é possível comparar os resultados obtidos. Todas as medições de tempo são feitas de acordo com os métodos disponibilizados pela classe *System* do Java. Através da chamada *System.currentTimemilis()* [javi] é possível obter o tempo actual do sistema em milissegundos. Como referido anteriormente, estes tempos são medidos aquando dos testes, através da classe *TestClass.java*. Para a medição e comparação dos tempos, a resolução até ao milissegundo é suficiente para este estudo.

6.2 Cenários de teste

Em seguida, iremos detalhar todos os testes efectuados, os resultados obtidos e retirar algumas notas sobre os assuntos considerados relevantes.

6.2.1 Controlo

Antes de avançarmos para os diversos testes que preparámos para testar as funcionalidades desenvolvidas, definimos um conjunto de testes a que chamámos de controlo, onde fazemos várias medições de tempos e configurações iniciais. Para alguns dos testes de controlo, testámos a diferença entre executar a ligação com o Socket Java normal, com o *NChannelSocket* (com e sem o mecanismo de mudança activo) e, num caso em particular, comparando ainda com o *WGet* [ubue]. Estas alternativas foram definidas para permitir perceber o *overhead* que todo o mecanismo de sonda e escolha de caminhos insere no sistema. Com as comparações com o *WGet* pretendemos analisar como se comporta a nossa implementação e a de Socket Java versus uma aplicação fora do contexto Java. O *WGet* foi escolhido por ser uma aplicação que se encontra nos sistemas Linux, e que tem uma boa *performance* e boas funcionalidades de extracção de ficheiros via web (HTTP, FTP, entre outros).

6.2.1.1 Teste ao tamanho do bloco

Com o primeiro teste de controlo pretendemos encontrar um tamanho aceitável para o tamanho do bloco, i.e. o tamanho de cada pedido, em termos de número de bytes. Analisámos cinco valores: 32KB (32768 bytes), 128KB (131072 bytes), 512KB (524288 bytes), 1MB (1048576 bytes) e 20MB (o ficheiro de 20MB na totalidade, num só pedido). Os resultados estão presentes nas Tabelas 6.2, 6.3, 6.4, 6.5 e 6.6, respectivamente. Para este teste, utilizamos a ligação com a velocidade máxima disponível no nosso primeiro ambiente de teste, i.e. o canal-12. Os valores testados permitem observar o peso deste parâmetro no tempo total de transferência com o nosso sistema (*NChannelSocket*) comparando com

uma aplicação de *Socket Java* comum (sem todo o mecanismo de histórico, e comutação de canais).

Tabela 6.2: Tamanho do bloco de 32768 bytes

Teste	Tempo total Socket Java (s)	Tempo total *NChannel-Socket* (s)	Tempo total *NChannel-Socket* sem *switch* (s)
1	196,609	197,307	196,076
2	198,409	192,519	199,181
3	199,473	196,536	194,756
4	202,419	198,126	199,395
5	198,057	200,443	201,022
6	196,960	198,212	196,772
7	197,061	201,656	192,633
8	198,028	197,148	197,746
9	193,705	198,157	197,060
10	193,943	203,229	195,126
Média	**197,466**	**198,333**	**196,977**
Mínimo	**193,705**	**192,519**	**192,633**
Máximo	**202,419**	**203,229**	**201,022**
Desvio Padrão	**2,535**	**2,964**	**2,489**

Tabela 6.3: Tamanho do bloco de 131072 bytes

Execução	Tempo total Socket Java (s)	Tempo total *NChannel-Socket* (s)	Tempo total *NChannel-Socket* sem *switch* (s)
1	172,833	173,680	173,693
2	172,351	173,899	177,255
3	173,722	173,802	176,124
4	171,765	173,320	172,587
5	177,000	175,581	175,083
6	172,289	175,442	175,833
7	173,601	174,069	174,978
8	175,688	175,337	174,230
9	173,808	174,188	174,281
10	174,382	173,099	175,528
Média	**173,744**	**174,242**	**174,959**
Mínimo	**171,765**	**173,099**	**172,587**
Máximo	**177,000**	**175,581**	**177,255**
Desvio Padrão	**1,620**	**0,897**	**1,331**

O valor do tamanho do bloco de 32KB foi escolhido porque é um valor normalmente definido como o tamanho do *buffer* de transmissão do TCP [tcp, sys, TDG], e suscitou alguma curiosidade ver se teria algum impacto. Contudo, as actuais implementações dos sistemas operativos possuem tamanhos das janelas superiores ao valor testado (no sistema *Ubuntu* utilizado temos um valor por omissão de 112KB para os *buffers* de envio e recepção de um socket TCP [ubud]), sendo que este acabou por demonstrar uma pior *performance* do que os restantes valores. Os outros valores foram simples aumentos para testar a influência do tamanho no tempo de transferência. O último dos valores testado

Tabela 6.4: Tamanho do bloco de 524288 bytes

Execução	Tempo total Socket Java (s)	Tempo total *NChannel-Socket* (s)	Tempo total *NChannel-Socket* sem *switch* (s)
1	167,547	166,824	167,351
2	167,739	168,262	168,026
3	167,094	167,581	167,601
4	167,129	167,631	166,664
5	166,910	167,519	167,308
6	167,050	167,484	167,729
7	166,705	169,569	167,095
8	167,123	168,732	166,828
9	168,407	167,724	168,409
10	166,887	167,814	167,159
Média	**167,259**	**167,914**	**167,417**
Mínimo	**166,705**	**166,824**	**166,664**
Máximo	**168,407**	**169,569**	**168,409**
Desvio Padrão	**0,506**	**0,767**	**0,536**

Tabela 6.5: Tamanho do bloco de 1048576 bytes

Execução	Tempo total Socket Java (s)	Tempo total *NChannel-Socket* (s)	Tempo total *NChannel-Socket* sem *switch* (s)
1	167,406	165,871	166,098
2	166,121	167,306	166,978
3	166,587	166,412	167,030
4	166,324	166,701	167,000
5	165,790	167,542	165,944
6	167,269	166,706	166,608
7	167,704	166,477	166,161
8	167,024	166,265	166,788
9	166,060	165,766	166,189
10	165,832	166,984	166,476
Média	**166,612**	**166,603**	**166,527**
Mínimo	**165,790**	**165,766**	**165,944**
Máximo	**167,704**	**167,542**	**167,030**
Desvio Padrão	**0,694**	**0,572**	**0,412**

Tabela 6.6: Tamanho do bloco de 20 MB - Totalidade do ficheiro um só pedido

Execução	Tempo total Socket Java (s)	Tempo total *NChannelSocket* (s)	Tempo total *NChannelSocket* sem *switch* (s)	*WGet* (s)
1	165,374	165,648	165,782	165
2	165,194	166,055	165,189	166
3	165,703	166,109	165,027	164
4	165,499	165,412	165,366	166
5	166,044	166,568	165,646	165
6	165,064	164,896	166,233	166
7	165,286	165,900	165,226	164
8	165,492	165,545	165,744	166
9	166,421	166,994	165,912	165
10	165,885	165,246	165,893	166
Média	**165,596**	**165,837**	**165,602**	**165,300**
Mínimo	**165,064**	**164,896**	**165,027**	**164,000**
Máximo	**166,421**	**166,994**	**166,233**	**166,000**
Desvio Padrão	**0,420**	**0,627**	**0,385**	**0,823**

(o tamanho total do ficheiro) é relevante para comparar o tempo que leva a transferir o ficheiro com o *NChannelSocket*, com um Socket Java e com o *WGet*. Estes testes permitem determinar o número de pedidos ao servidor, até o ficheiro estar totalmente transferido. O número aproximado de pedidos ao servidor e o tempo que cada pedido demora a transferir, conforme o tamanho do bloco utilizado, encontra-se na Tabela 6.7. Através destes números, juntamente com os valores obtidos, permite-nos estabelecer uma relação directa entre o tempo que demora a transferência e o tamanho do bloco utilizado. Um tamanho do bloco mais pequeno implica um maior número de pedidos ao servidor e, devido à nossa implementação, implica um número maior de testes de mudança de conexão (um teste de mudança de conexão por cada pedido ao servidor), a um ritmo muito elevado (250ms para o tamanho do bloco de 32KB).

Tabela 6.7: Número de pedidos ao servidor por tamanho do bloco

Tamanho do bloco (bytes)	Número de pedidos	Periodicidade de teste de alternativas de conexão (s)
32768	640	0.25
131072	160	1
524288	40	4
1048576	20	8
20MB	1	160

Depois deste teste, definimos o tamanho do bloco a utilizar nos restantes testes para o valor de 524288 bytes, pois foi aquele que nos apresentou um melhor tempo de transferência mantendo, para este caso em particular, um número de acessos ao servidor razoável para conseguir interpretar e mudar de conexão se necessário. Assim, no resto do capítulo, a não ser que seja explicitado o contrário, o valor utilizado para cada pedido é

de 512KB (524288 bytes).

6.2.1.2 Testes de controlo individuais

Os próximos testes serviram para medir quanto tempo leva a transferência a ser efectuada, usando cada um dos canais configurados, servindo como uma base para o resto das comparações. Cada ligação está configurada como indicado na Tabela 6.1. Nestes testes apenas é utilizado o respectivo canal, e fizemos também comparações entre transferir com o mecanismo de mudança ligado e desligado e com o Java Socket normal. Nas Tabelas 6.8 e 6.9 encontram-se os resultados para o teste ao pior canal (canal-10) e ao canal intermédio (canal-11), respectivamente. Os resultados equivalentes para o melhor canal (canal-12) encontram-se nos testes anteriores de tamanho do bloco (Subsecção 6.2.1.1), mais propriamente do teste aos 512KB (Tabela 6.4).

Tabela 6.8: Controlo usando pior canal

Execução	Tempo total Socket Java (s)	Tempo total *NChannelSocket* (s)	Tempo total *NChannelSocket* sem *switch* (s)
1	245,032	243,561	247,337
2	248,330	244,351	241,387
3	246,563	243,513	247,821
4	244,212	244,576	244,774
5	245,528	246,816	243,327
6	245,341	244,912	245,514
7	244,551	244,985	247,470
8	247,429	244,441	242,694
9	241,573	247,025	248,550
10	247,990	248,830	245,229
Média	**245,655**	**245,301**	**245,410**
Mínimo	**241,573**	**243,513**	**241,387**
Máximo	**248,330**	**248,830**	**248,550**
Desvio Padrão	**2,029**	**1,712**	**2,403**

Para facilitar a compreensão e mostrar que de facto cada um dos canal comporta-se de forma diferente, consoante as suas características, juntamos os resultados dos três canais usando o *NChannelSocket* na Tabela 6.10. Aqui podemos observar mais facilmente que o canal-10 (pior) tem piores resultados, o canal-11 tem resultados melhores que o pior e, como era esperado, o canal-12 tem os melhores resultados entre os três canais disponíveis. Isto mostra que as configurações dos canais e todo o modelo de teste está equilibrado e permite modelar o que é pretendido.

6.2.1.3 Teste ao *overhead* da comutação e sonda de canais

Neste teste pretendemos descobrir se o mecanismo de criação/gestão de novas sondas introduziria ou não um *overhead* (sobrecarga) significativo no tempo total de transferência.

Tabela 6.9: Controlo usando canal intermédio

Execução	Tempo total Socket Java (s)	Tempo total *NChannelSocket* (s)	Tempo total *NChannelSocket* sem *switch* (s)
1	201,939	198,185	194,385
2	194,988	195,149	192,196
3	194,190	199,795	195,759
4	202,547	195,729	194,757
5	200,249	196,028	194,080
6	192,788	199,790	190,800
7	200,601	200,458	193,491
8	203,430	199,785	198,936
9	198,046	190,045	201,818
10	200,037	203,136	198,481
Média	**198,882**	**197,810**	**195,470**
Mínimo	**192,788**	**190,045**	**190,800**
Máximo	**203,430**	**203,136**	**201,818**
Desvio Padrão	**3,721**	**3,684**	**3,359**

Tabela 6.10: Controlo individual de cada canal do modelo 1 - usando *NChannelSocket*

Execução	Tempo total canal-10 (s)	Tempo total canal-11 (s)	Tempo total canal-12 (s)
1	243,561	198,185	166,824
2	244,351	195,149	168,262
3	243,513	199,795	167,581
4	244,576	195,729	167,631
5	246,816	196,028	167,519
6	244,912	199,790	167,484
7	244,985	200,458	169,569
8	244,441	199,785	168,732
9	247,025	190,045	167,724
10	248,830	203,136	167,814
Média	**245,301**	**197,810**	**167,914**
Mínimo	**243,513**	**190,045**	**166,824**
Máximo	**248,830**	**203,136**	**169,569**
Desvio Padrão	**1,712**	**3,684**	**0,767**

Este teste foi feito comparando o tempo de execução do *NChannelSocket* (com o mecanismo de mudança de conexão e de criação de sondas activo), com o tempo da transferência utilizando Java Socket comum. Os resultados estão apresentados na Tabela 6.11.

Tabela 6.11: Overhead do mecanismo de sondas

Execução	Tempo total Java Socket (s)	Tempo total *NChannel-Socket* (s)
1	167,547	166,952
2	167,739	166,981
3	167,094	167,431
4	167,129	167,345
5	166,910	167,501
6	167,050	167,459
7	166,705	167,278
8	167,123	167,075
9	168,407	167,299
10	166,887	167,820
Média	**167,259**	**167,314**
Mínimo	**166,705**	**166,952**
Máximo	**168,407**	**167,820**
Desvio Padrão	**0,506**	**0,264**

Como podemos observar, a execução de todo o mecanismo, em média, apenas implica milésimos de segundos de sobrecarga sobre o tempo de transferência, em relação à alternativa do Java Socket. Neste teste a aplicação usando *NChannelSocket* arrancava a partir do melhor canal, e lá se manteve, pelo que as sondas criadas para os outros dois caminhos disponíveis apenas estavam, neste caso, a sobrecarregar. Os tempos com *Java Socket* também foram medidos utilizando o canal-12.

6.2.2 Teste 1 - Comutação sem histórico

Neste primeiro teste pretende-se verificar se a aplicação, arrancando sem qualquer informação prévia das conexões existentes, e partindo do suposto que não escolhe como inicial a melhor das conexões disponíveis (canal-12), consegue alcançar alguma conexão com melhores características e, em caso positivo, quanto tempo demora, e se se mantém nessa conexão dita melhor. Os resultados deste teste encontram-se na Tabela 6.12.

Através da interpretação dos resultados conseguimos observar que sempre que a conexão partiu de um endereço pior conseguiu, ao fim de algum tempo, comutar para a melhor conexão da altura (que foi sempre o canal-12). O método de mudança foi sempre o tipo aleatório, pois através das sondas (*probes*) a aplicação conseguiu descobrir canais com possibilidades para ser melhor que o canal em uso, i.e. com melhor *SmothedRTT* / RTO. Os tempos de transferência mantém-se relativamente estáveis (com uma pequena variância). O tempo assinalado na última coluna indica o momento após o início da transferência onde a comutação ocorreu. Este tempo está relacionado directamente com a comutação do tipo aleatório e acaba por não ser totalmente estável. Existem vezes em

Tabela 6.12: Teste 1 - Comutação sem histórico

Execução	Tempo total(s)	Escolheu o melhor?	Tempo depois do início onde a alteração ocorreu (s)
1	172,506	Sim	13,855
2	169,731	Sim	6,413
3	168,987	Sim	6,511
4	171,513	Sim	12,869
5	169,671	Sim	6,674
6	172,765	Sim	8,338
7	171,669	Sim	7,898
8	169,012	Sim	6,458
9	171,220	Sim	7,780
10	171,011	Sim	6,603
11	170,961	Sim	6,441
12	172,116	Sim	13,299
13	171,724	Sim	12,715
14	171,296	Sim	6,715
15	169,502	Sim	6,958
16	170,355	Sim	6,411
17	172,067	Sim	13,523
18	171,344	Sim	12,737
19	169,350	Sim	6,411
20	174,018	Sim	19,508
Média	**171,041**		**9,406**
Mínimo	**168,987**		**6,411**
Máximo	**174,018**		**19,508**
Desvio Padrão	**1,359**		**3,807**

que 6 segundos bastam para a comutação aleatória ser activada, outras em que chegam ao 13 segundos. Da maneira como o mecanismo está implementado, não temos outro modo de controlar esta aleatoriedade. No Capítulo 7 voltamos a referenciar este assunto.

6.2.3 Teste 2 - Comutação com histórico

No segundo teste parte-se do ficheiro de histórico das conexões efectuadas, e testa-se se o *NChannelSocket* consegue seleccionar logo a melhor das conexões disponíveis e se se irá manter nessa conexão durante toda a transferência. Os resultados correspondentes a este teste encontram-se na Tabela 6.13.

Tabela 6.13: Teste 2 - Comutação com histórico

Execução	Tempo total(s)	Escolheu o melhor?	Ficou no melhor?
1	168,500	Sim	Sim
2	166,956	Sim	Sim
3	167,703	Sim	Sim
4	167,253	Sim	Sim
5	167,706	Sim	Sim
6	169,569	Sim	Sim
7	166,518	Sim	Sim
8	168,451	Sim	Sim
9	167,793	Sim	Sim
10	167,827	Sim	Sim
11	169,567	Sim	Sim
12	166,546	Sim	Sim
13	166,740	Sim	Sim
14	167,243	Sim	Sim
15	167,102	Sim	Sim
16	167,562	Sim	Sim
17	167,599	Sim	Sim
18	167,811	Sim	Sim
19	166,965	Sim	Sim
20	168,161	Sim	Sim
Média	**167,679**		
Mínimo	**166,518**		
Máximo	**169,569**		
Desvio Padrão	**0,857**		

Comparando o Teste 1 (Subsecção 6.2.2) com este segundo teste, conseguimos observar que o tempo que demora a transferência é reduzido significativamente. Isto deve-se ao facto de neste segundo teste o sistema arrancar sempre pela melhor das conexões disponíveis, mantendo-se nessa conexão. De notar que neste teste não pretendemos comparar quando algo corre mal, mas simplesmente se, arrancando de um histórico, a *performance* global em termos de tempo de transferência melhorava, o que de facto acontece.

6.2.4 Teste 3 - O melhor canal fica indisponível

Neste teste pretendemos analisar como se comporta o sistema caso um dos canais, neste caso o melhor deles (canal-12), fique indisponível. O teste foi concebido para analisar como o sistema reage em caso de falha repentina e persistente. Caso o canal em uso fique indisponível, quanto tempo demora o sistema a reagir, e ao mudar, se muda para o melhor endereço dos disponíveis. 20 segundos após o arranque da transferência, metemos o melhor canal indisponível através do *ipfw*, bloqueando qualquer pacote IP de e para aquele endereço. É como se a conexão tivesse ido abaixo.

Outro detalhe importante a salientar neste teste é a variação dos valores de *timeout* (de conectividade e de leitura). Para os testes até aqui, estes *timeouts* eram pouco relevantes, pois o ambiente é controlado e falhas de conexão são raras. Contudo, nos próximos testes (Testes 3 a 5) este valor é importante, pois forçamos situações em que os canais falham ou pioram, e a probabilidade de algum dos valores de *timeout* acontecer é muito elevada. Para testar o melhor valor de *timeout* e a sua importância no comportamento do sistema e no tempo total de transferência, repetimos este teste variando este valor. Os valores de *timeout* testados foram de 500ms, 1s e 5s, sendo que os resultados estão apresentados nas Tabelas 6.14, 6.15 e 6.16, respectivamente.

Neste teste o sistema arranca com histórico, sendo que a ligação escolhida como sendo a inicial é sempre a melhor disponível (como acontece para o Teste 2 - Secção 6.2.3). A diferença de 20 segundos (20000ms) em média entre o início da transferência e o canal ir abaixo deve-se à configuração, como explicado anteriormente. Em cada um dos testes, o valor de parâmetro de *timeout* para o mesmo está bem presente na última coluna. A diferença de tempo entre o canal falhar e o sistema dar pela falha está relacionada directamente com o valor definido de *timeout*. Em cada um dos casos, findo o tempo definido, a aplicação lança uma excepção de leitura, pois o socket está à espera de ler algum byte, mas acaba por não receber nada durante o tempo estipulado. O sistema acaba por fechar a ligação que falhou, abrindo outra que considerar a melhor alternativa das ainda disponíveis, prosseguindo com a transferência.

Analisando os resultados obtidos conseguimos verificar que existe uma relação entre o tempo médio de transferência e o valor definido para o *timeout*. Contudo, para este primeiro ambiente de teste, os resultados obtidos não são os que seriam de esperar: que com um valor de *timeout* mais pequeno, a ligação daria pelo erro e comutava mais rapidamente, logo terminaria mais rápido. Verifica-se o oposto, que com um valor de *timeout* maior, a ligação fica mais estável e, apesar de demorar mais tempo a dar pela falha, consegue terminar (em média) mais rapidamente. O que se verificou foi que, neste ambiente de teste em particular, um valor de *timeout* muito pequeno $< 1s$ provocou grande instabilidade na transferência. Ou seja, a ligação várias vezes que não conseguia conectar-se aos servidores a tempo, sendo que começavam a ser registados *timeouts* de conectividade. Devido ao funcionamento do nosso sistema aquando uma detecção de uma falha, este procura outro caminho disponível e tenta conectar-se. Um caso que se verificou algumas

Tabela 6.14: Teste 3 - Melhor canal fica indisponível, *timeout* 500ms

Execução	Tempo total(s)	Diff Tempo entre o inicio e o canal ir abaixo (s)	Diff entre a falha e a comutação (s)
1	198,054	20,009	0,595
2	196,569	20,006	0,657
3	194,237	20,005	0,695
4	233,872	20,006	0,637
5	227,055	20,010	0,654
6	197,684	20,070	0,696
7	196,339	20,006	0,667
8	200,968	20,006	0,648
9	191,706	20,112	0,662
10	203,770	20,005	0,678
11	212,944	20,079	0,687
12	198,731	20,116	0,646
13	212,895	20,123	0,655
14	209,373	20,102	0,885
15	188,813	20,298	0,588
16	191,042	20,006	0,704
17	216,850	20,233	0,717
18	235,872	20,008	0,554
19	213,099	20,006	193,093
20	218,457	20,005	0,718
Média	**206,917**		**10,292**
Mínimo	**188,813**		**0,554**
Máximo	**235,872**		**193,093**
Desvio Padrão	**14,115**		**43,027**

Tabela 6.15: Teste 3 - Melhor canal fica indisponível, *timeout* 1s

Execução	Tempo total(s)	Diff Tempo entre o inicio e o canal ir abaixo (s)	Diff entre a falha e a comutação (s)
1	201,699	20,011	1,127
2	194,460	20,006	1,162
3	195,687	20,006	1,165
4	197,658	20,006	1,152
5	203,265	20,014	1,174
6	193,493	20,017	1,187
7	198,215	20,030	1,135
8	232,812	20,020	1,159
9	199,209	20,006	1,114
10	194,768	20,047	1,185
11	195,710	20,048	1,120
12	238,032	20,050	2,193
13	198,306	20,189	1,234
14	194,096	20,209	1,170
15	199,328	20,252	1,216
16	194,807	20,149	1,172
17	193,480	20,164	1,115
18	236,032	20,159	2,306
19	193,436	20,206	1,094
20	195,815	20,323	1,126
Média	**202,515**		**1,265**
Mínimo	**193,436**		**1,094**
Máximo	**238,032**		**2,306**
Desvio Padrão	**14,552**		**0,339**

Tabela 6.16: Teste 3 - Melhor canal fica indisponível, *timeout* 5s

Execução	Tempo total(s)	Diff Tempo entre o inicio e o canal ir abaixo (s)	Diff entre a falha e a comutação (s)
1	209,401	20,014	5,114
2	205,318	20,007	5,107
3	213,477	20,010	5,124
4	220,150	20,006	5,229
5	193,366	20,009	5,129
6	198,953	20,006	5,227
7	195,215	20,006	5,158
8	198,199	20,006	5,215
9	195,216	20,006	5,148
10	194,500	20,006	5,183
11	200,434	20,006	5,162
12	202,054	20,005	5,172
13	205,790	20,050	5,110
14	195,263	20,089	5,365
15	200,619	20,120	5,138
16	200,730	20,011	5,102
17	198,339	20,006	5,167
18	199,306	20,005	5,104
19	198,485	20,006	5,066
20	200,536	20,026	5,102
Média	**201,268**		**5,156**
Mínimo	**193,366**		**5,066**
Máximo	**220,150**		**5,365**
Desvio Padrão	**6,742**		**0,066**

vezes foi que, quando o canal-12 falhava, os 500ms definidos de valor de *timeout* não chegavam para outra conexão ser estabelecida, o que provocava uma excepção por *timeout*, e uma nova escolha de outra conexão, entrando num ciclo de tentativas de conexão e lançamentos de excepções de *timeout*, até finalmente algum dos servidores conseguir aceitar a ligação. Todas estas mudanças, mesmo que pequenas, provocam um aumento do tempo que o sistema demora a estabilizar e, em último caso, um aumento do tempo total da transferência. Como será explicado mais à frente no Teste 5 (Secção 6.2.6) este problema nem sempre acontece. Não conseguimos perceber atempadamente se este atraso de tentativa de conectividade é introduzido pelo mecanismo de controlo de fluxo dos canais do *ipfw* ou se por algum atraso da rede interna do departamento onde os testes foram efectuados. De qualquer modo, conseguimos perceber que o valor de *timeout* em casos de falha ou problemas nos canais se torna relevante no comportamento do sistema, sendo que um valor de *timeout* mais pequeno pode ser vantajoso nuns casos e piorar o sistema noutros casos. Este aspecto será discutido novamente no capítulo seguinte (Capítulo 7).

Para efeitos de compreensão do funcionamento do *framework* de monitorização, apresentamos na Figura 6.4 um gráfico que representa uma das transferências do ficheiro. No eixo dos x temos o tempo em segundos e no eixo dos y temos o *throughput* da conexão em Bytes por segundo. De notar que a melhor conexão (canal-12) é a escolhida inicialmente e tem a capacidade máxima de 1MBit/s, antes de ir abaixo. Após esta falhar, o *NChannelSocket* comuta para a que considera a próxima melhor, que é o canal-11. Este canal mantém-se estável durante um certo intervalo de tempo mas, devido a problemas de medições com o valor de RTT medido, acaba por acontecer uma mudança aleatória para o canal-10 (que na altura possuía um melhor *SmoothedRTT* que o canal-11). Após comutar para o canal-10, o *NChannelSocket* continuou nesse canal até terminar a transferência, à velocidade máxima possível de 700KBit/s. Este gráfico representa da melhor maneira possível de que o sistema funciona e comuta de canal sempre que necessário.

6.2.5 Teste 4 - O melhor canal fica com pior *performance*

Com este quarto teste pretendemos verificar como o sistema actua em caso de quebra de *performance* das conexões. 20 segundos após o inicio da transferência, as propriedades da melhor conexão são alteradas (utilizando mais uma vez o *ipfw*) para os parâmetros apresentados na Tabela 6.17, em que o único parâmetro alterado foi o da capacidade. Deste modo, a conexão piora mas continua a funcionar. Pretende-se analisar como o sistema reage, quanto tempo leva a reagir e, se mudar, se o fará para a conexão adequada.

Tabela 6.17: Novas características da conexão melhor

Endereço	Capacidade (Kbps)	Atraso (RTT/2) (ms)	Tamanho da fila (entradas)	Taxa de perca de pacotes
canal-12	100	10	15	0.01

Pretende-se ainda analisar se este tempo de reacção muda consoante o número de

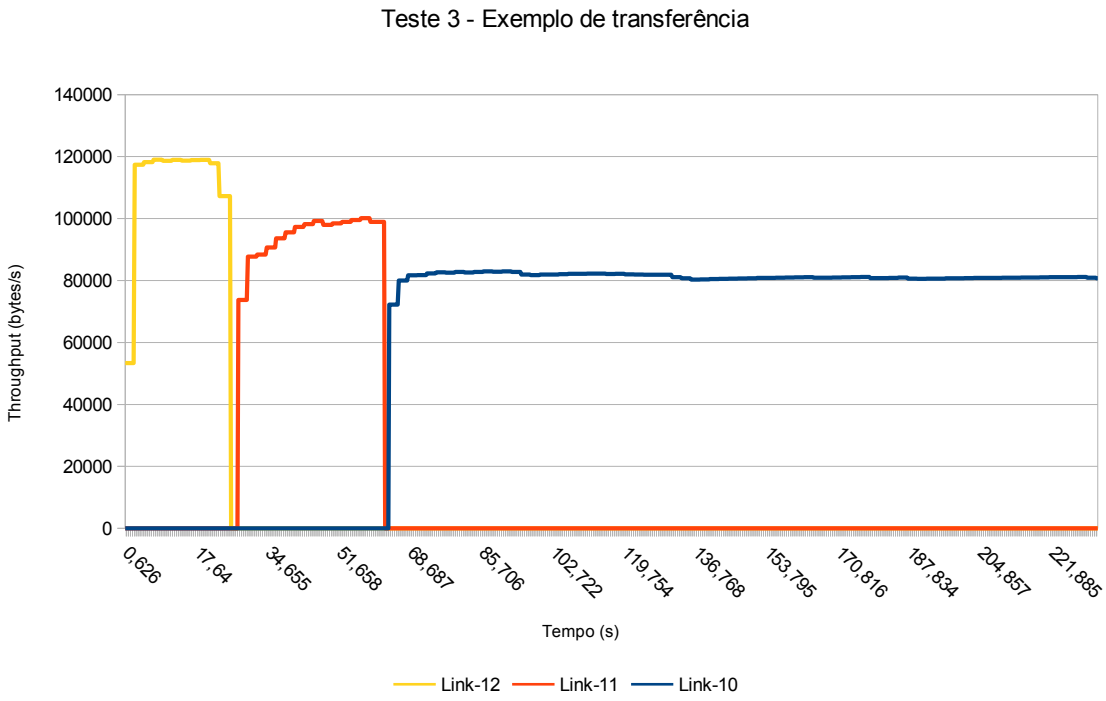

Figura 6.4: Teste 3 - Exemplo de transferência

pedidos ao servidor, i.e. o tamanho do tamanho do bloco. Testámos dois tamanhos, o utilizado normalmente (512KB) e um mais pequeno (128KB). Os resultados estão presentes nas Tabelas 6.18 e 6.19, respectivamente.

Estes testes reflectem que, mesmo que o canal gradualmente piore, a mudança para um canal considerado melhor acaba sempre por acontecer. Em especial em 3 ocasiões a mudança não chegou a ser detectada antes da conexão ficar tão má que acabou por dar uma excepção de *timeout* de leitura. Nesses casos, a mudança ocorreu, como no Teste 3 (Subsecção 6.2.4), i.e. como se de uma falha se tratasse. Apesar de termos um modo de testar, através do *throughput* ou do rácio de perca de pacotes, se uma conexão piorou, este método raramente acontece antes de um caso em que a mudança aleatória entra em acção. Apesar da ligação piorar consideravelmente, em termos de *throughput*, como trabalhamos comparando o valor recente ao do passado, a influência na média acaba por não ser significativa o suficiente para os valores que pretendemos comparar, i.e. o declínio de *performance* não é abrupto o suficiente para ter um impacto imediato na média, tendo sim um impacto gradual que acaba por nunca se tornar critico ao ponto de sugerir a mudança. Este facto é tanto mais visível conforme o tamanho do bloco utilizado. Comparando os valores obtidos na Tabela 6.18 (teste de 512KB) com a Tabela 6.19 (teste de 128KB) percebemos o que foi dito anteriormente. Apesar do tamanho do bloco de 128KB ser um número mais pequeno e, por isso, implicar que o teste de mudança de conexão ocorre num intervalo de tempo mais pequeno do que com 512KB, a transferência do

Tabela 6.18: Teste 4 - Melhor canal piora, tamanho do bloco de 512KB

Execução	Tempo total(s)	Aconselha mudar?	Diff tempo entre o inicio e ficar pior (s)	Diff tempo entre ficar pior e mudar (s)
1	338,468	Random	20,012	162,703
2	304,637	Random	20,008	118,462
3	325,965	Excepção	20,009	152,350
4	341,351	Random	20,009	165,377
5	380,187	Random	20,008	210,836
6	337,469	Random	20,020	161,680
7	415,719	Random	20,010	250,934
8	347,097	Random	20,009	168,850
9	341,405	Random	20,126	165,021
10	423,816	Random	20,151	255,100
11	409,098	Random	20,048	249,328
12	369,449	Random	20,157	200,609
13	328,996	Random	20,160	154,753
14	482,979	Random	20,094	330,944
15	416,170	Random	20,134	252,430
16	297,848	Random	20,138	117,720
17	300,728	Random	20,170	116,146
18	284,340	Excepção	20,268	106,796
19	350,756	Random	20,195	131,204
20	316,511	Excepção	20,278	96,446
Média	**355,649**			**178,384**
Mínimo	**284,340**			**96,446**
Máximo	**482,979**			**330,944**
Desvio Padrão	**51,339**			**62,364**

Tabela 6.19: Teste 4 - Melhor canal piora, tamanho do bloco de 128KB

Execução	Tempo total(s)	Aconselha mudar?	Diff tempo entre o inicio e ficar pior (s)	Diff tempo entre ficar pior e mudar (s)
1	382,983	Random	20,227	153,019
2	392,001	Random	20,009	146,525
3	365,507	Excepção	20,008	132,572
4	394,223	Random	20,009	163,955
5	408,294	Random	20,009	165,386
6	550,567	Random	20,025	386,908
7	467,096	Excepção	20,047	0,815
8	334,358	Random	20,155	124,298
9	382,206	Random	20,009	144,672
10	374,180	Random	20,009	141,411
11	399,664	Random	20,040	169,358
12	387,770	Random	20,009	152,669
13	371,073	Random	20,009	148,620
14	345,063	Random	20,154	109,046
15	368,272	Random	20,009	169,408
16	345,813	Random	20,009	153,870
17	388,962	Random	20,038	166,376
18	384,756	Random	20,034	160,295
19	384,465	Random	20,009	151,014
20	373,979	Random	20,008	142,506
Média	**390,062**			**154,136**
Mínimo	**334,358**			**0,815**
Máximo	**550,567**			**386,908**
Desvio Padrão	**46,654**			**65,872**

ficheiro acaba por demorar mais tempo, muito devido ao tamanho do tamanho do bloco e à influência que este tem sobre o tempo total de transferência (Subsecção 6.2.1.1).

Este sistema (de teste à *performance* de uma conexão, Subsecção 5.3.6) já foi, porém, detectado a funcionar, noutras ocasiões, quando o ficheiro de histórico que tínhamos era antigo, sendo que os valores para uma dada conexão eram agora mais baixos. Ao comparar a média do que costumava acontecer nessa conexão (o passado/histórico da mesma) com o novo valor, verificava que as características tinham piorado significativamente, sugerindo a mudança para uma conexão melhor, caso existisse. Este caso indica que seria necessário uma forma mais afinada de ajustar estes parâmetros de comparação entre o presente e o passado. Ficam como referência a testes futuros no Capítulo 7.

Como aconteceu no Teste 3 (Subsecção 6.2.4), voltamos a mostrar um exemplo do funcionamento do *NChannelSocket* para este quarto teste na Figura 6.5. Através do gráfico observamos que, por volta dos 20 segundos a melhor conexão sofre a mudança de parâmetros, passando da sua velocidade máxima (1MBit/s) para uma descida suave do *throughput*, até atingir o valor de 100KBit/s configurado. Quando a conexão fica má é quando por fim o sistema decide que vale a pena comutar de ligação, escolhendo a que estaria em melhor estado, neste caso o canal-10. Manteve-se nesse canal até ao fim, na velocidade máxima configurada (700KBit/s).

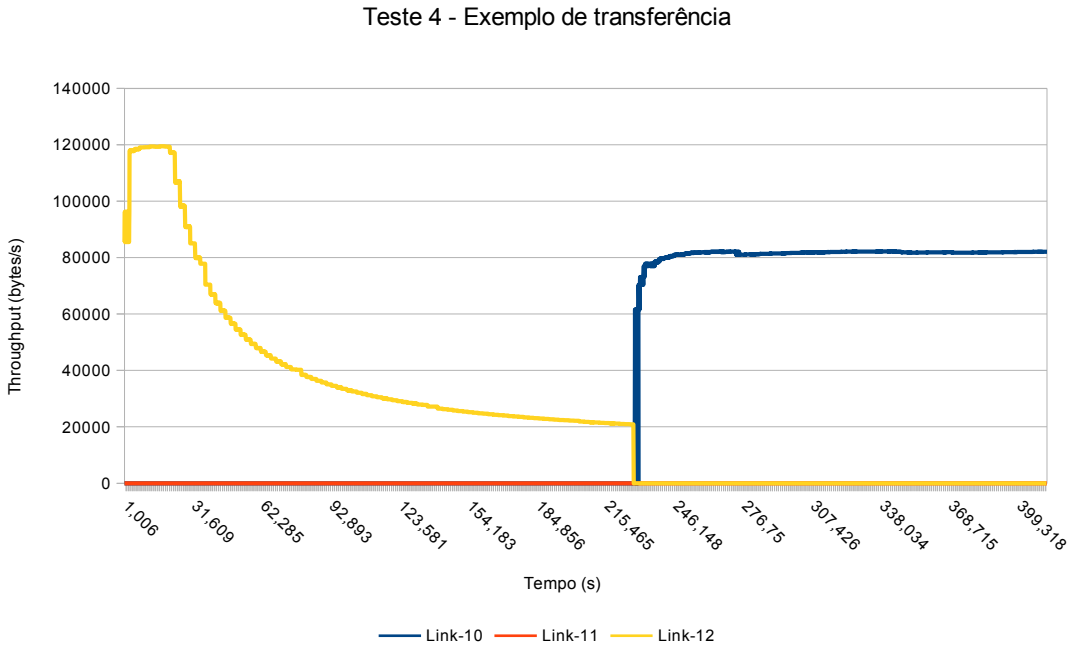

Figura 6.5: Teste 4 - Exemplo de transferência

6.2.6 Teste 5 - Teste de duas interfaces na origem

Este último teste é onde utilizamos o segundo ambiente de teste. Primeiro fizemos alguns testes de controlo específicos para este caso (Subsecção 6.2.6.1), sendo que posteriormente fazemos o teste pretendido (Subsecção 6.2.6.2).

6.2.6.1 Testes de controlo

Como estamos perante um ambiente de teste diferente, pretendemos ver como se comporta o sistema utilizando somente a ligação por fio (*Ethernet*), e utilizando a ligação sem fios (*Wireless*). Os resultados estão presentes na Tabela 6.20. Para estes testes de controlo o tamanho do bloco foi definido como sendo 512KB e os *timeouts* foram definidos com o valor de 500ms.

Tabela 6.20: Teste 5 - Controlo

Execução	Tempo total Wireless(s)	Tempo total Wired (s)
1	124,125	117,478
2	121,390	116,055
3	119,099	115,556
4	117,776	116,781
5	117,261	115,181
6	118,548	116,221
7	118,170	116,929
8	117,732	115,767
9	119,541	116,112
10	122,456	115,718
11	124,865	115,500
12	117,625	116,019
13	120,021	116,028
14	118,332	115,604
15	117,231	115,767
16	118,604	115,228
17	117,731	115,537
18	118,164	115,950
19	121,383	115,726
20	119,344	115,508
Média	**119,470**	**115,933**
Mínimo	**117,231**	**115,181**
Máximo	**124,865**	**117,478**
Desvio Padrão	**2,248**	**0,572**

Através dos dados podemos observar que, como seria espectável, a ligação somente por *Wireless* tem uma *performance* inferior à ligação por cabo (*Ethernet*). Não só o *Wireless* demora mais tempo a fazer a transferência, mas também se nota que esse tempo tem uma maior variação (o desvio padrão é maior), apesar de pequeno. Este teste serve de ponto de controlo para o teste seguinte.

6.2.6.2 O canal Ethernet fica indisponível

Por fim, neste teste pretendemos observar como o sistema reage quando é na origem que a falha ocorre e, mais propriamente, quando temos duas interfaces, em formatos diferentes. Neste caso, provocando uma falha no canal por fio, pretendemos ver se a aplicação comuta para o canal *Wireless*, quanto tempo demora a detectar a falha e a comutar de ligação.

Também achámos interessante verificar como se comportava a comutação conforme o *timeout* parametrizado. Assim, testámos definindo os *timeouts* com valores de 500ms, 1s e 5s. Os resultados estão presentes na Tabelas 6.21, 6.22 e 6.23.

Tabela 6.21: Teste 5 - canal Ethernet fica indisponível, *timeout* 500ms

Execução	Tempo total(s)	Diff entre o inicio e o canal ir abaixo (s)	Diff entre o canal ir abaixo e mudar (s)
1	115,509	15,009	0,602
2	115,907	15,007	0,666
3	116,129	15,007	0,699
4	118,052	15,006	0,730
5	115,142	15,006	0,764
6	115,639	15,042	0,823
7	116,366	15,010	0,666
8	117,558	15,006	0,653
9	115,384	15,006	0,713
10	115,060	15,043	0,579
11	116,167	15,045	0,871
12	116,241	15,040	0,599
13	113,280	15,112	0,616
14	117,185	15,079	0,645
15	116,106	15,086	0,646
16	116,258	15,111	0,607
17	116,580	15,246	0,544
18	116,703	15,246	0,845
19	116,335	15,128	0,953
20	116,279	15,106	0,546
Média	**116,094**		**0,688**
Mínimo	**113,280**		**0,544**
Máximo	**118,052**		**0,953**
Desvio Padrão	**0,999**		**0,113**

Este teste permitiu perceber que, utilizando um *timeout* muito pequeno funcionou da forma que era esperada, quando comparado com que aconteceu no primeiro ambiente de teste (Subsecção 6.2.4). Aqui, quanto maior foi o *timeout*, mais tempo demorou a transferência a ser concluída, mesmo que a diferença não seja muito acentuada, os valores fazem sentido. A última coluna de cada uma das tabelas demonstra bem o tempo que o *timeout* demora a disparar e quanto tempo o sistema demora a comutar de conexão, sendo que o *timeout* influência directamente o tempo que o sistema demora a comutar.

Tabela 6.22: Teste 5 - canal Ethernet fica indisponível, *timeout* 1s

Execução	Tempo total(s)	Diff entre o inicio e o canal ir abaixo (s)	Diff entre o canal ir abaixo e mudar (s)
1	117,686	15,097	1,105
2	117,068	15,006	1,211
3	116,913	15,006	1,097
4	117,948	15,006	1,438
5	116,877	15,048	1,134
6	117,256	15,052	1,190
7	117,152	15,087	1,057
8	117,114	15,021	1,243
9	116,760	15,038	1,218
10	116,909	15,046	1,192
11	116,849	15,019	1,141
12	116,926	15,006	1,215
13	117,167	15,143	1,381
14	116,969	15,076	1,059
15	116,930	15,117	1,084
16	116,953	15,012	1,253
17	116,722	15,149	1,066
18	117,114	15,218	1,297
19	117,348	15,168	1,071
20	116,468	15,155	1,466
Média	**117,056**		**1,196**
Mínimo	**116,468**		**1,057**
Máximo	**117,948**		**1,466**
Desvio Padrão	**0,329**		**0,124**

Tabela 6.23: Teste 5 - canal Ethernet fica indisponível, *timeout* 5s

Execução	Tempo total(s)	Diff entre o inicio e o canal ir abaixo (s)	Diff entre o canal ir abaixo e mudar (s)
1	122,635	15,131	5,224
2	166,224	15,007	5,195
3	121,655	15,019	5,143
4	120,853	15,006	5,403
5	119,385	15,030	5,144
6	120,514	15,006	5,251
7	120,815	15,006	5,234
8	120,900	15,070	5,071
9	120,034	15,050	5,228
10	120,585	15,005	5,191
11	120,706	15,006	5,113
12	121,018	15,083	5,220
13	121,017	15,009	5,294
14	120,606	15,162	5,354
15	120,965	15,178	5,074
16	120,548	15,031	5,184
17	122,106	15,151	5,144
18	121,048	15,104	5,286
19	120,892	15,010	5,311
20	121,218	15,006	5,200
Média	**123,186**		**5,213**
Mínimo	**119,385**		**5,071**
Máximo	**166,224**		**5,403**
Desvio Padrão	**10,153**		**0,088**

6.3 Sumário

Neste capítulo apresentámos os testes que efectuámos, que se dividiram em dois ambientes distintos, cada um com o seu conjunto de testes.

Os testes de controlo (Subsecção 6.2.1) permitiram-nos avaliar como se comporta cada ligação dos ambientes de teste criados. Os testes 1 e 2 (Subsecção 6.2.2 e 6.2.3, respectivamente) permitiram-nos, com sucesso, observar os mecanismos de mudança de conexão em funcionamento. Os testes 3 (Subsecção 6.2.4) e 4 (Subsecção 6.2.5) permitiram-nos observar que o *NChannelSocket* comuta, com sucesso, para outras ligações em caso de falha ou perca de *performance*. Através do teste 5 (Subsecção 6.2.6) observámos que o sistema comuta com sucesso, não só quando existe mais que um endereço disponível no destino, mas também quando existe mais que um endereço de origem.

Conseguimos, portanto, verificar que o nosso padrão e *Framework* auxiliar funcionam correctamente, fazendo a comutação de canais como pretendido nos objectivos deste trabalho. Quando uma conexão piora ou quebra conseguimos comutar de ligação, tanto com vários endereços de destino e um de origem (primeiro ambiente), como com vários endereços de origem (segundo ambiente).

Conseguimos ver que os tempos obtidos, mesmo comparando com outras aplicações como o caso do *WGet*, ou com as versões mais simples de Java Socket, são bastantes semelhantes.

Os resultados obtidos neste capítulo permitem-nos chegar às conclusões anunciadas no capítulo seguinte.

7

Conclusões e trabalho futuro

Este trabalho teve como objectivo principal o de criar uma solução a nível aplicação que permitisse contornar possíveis problemas na rede, num ambiente onde a existência de mais que um caminho para um determinado destino é assumido.

O desenrolar deste trabalho ocorreu em duas fases distintas. Uma fase de pesquisa de material bibliográfico e levantamento de requisitos (Capítulos 2 e 3) e uma segunda fase de desenvolvimento do padrão de programação e do respectivo *framework* (*NChannelSocket*) (Capítulos 4 e 5). Após o desenvolvimento, foi possível passar à fase seguinte de teste e a avaliação de todo trabalho (Capítulo 6). Os testes foram divididos em dois modelos, em que num deles pretendemos simular 3 canais distintos para um certo objecto (vários endereços/interfaces no destino) (Testes 1 a 4, Subsecções 6.2.2 a 6.2.5) e no outro (Teste 5, Subsecção 6.2.6) que pretendemos simular várias interfaces na origem.

7.1 Conclusões

Da utilização e criação de aplicações com o padrão desenvolvido verificámos que é fácil criar uma aplicação para o *NChannelSocket*, desde que se saiba como funciona o Java Socket convencional. O padrão facilmente se torna intuitivo de utilizar e de fácil compreensão, sendo que o objectivo de simples usabilidade foi, a nosso ver, alcançado.

Da avaliação resultante dos testes concluímos que o *NChannelSocket* faz o que foi proposto fazer, i.e. arranja modos de mudar de canal na ocorrência de uma falha ou quando este piora. Tem ainda o beneficio de conseguir comutar de canal mesmo quando este está a funcionar correctamente, se algum canal considerado melhor for descoberto. Consegue comutar de conexões TCP, sondar outras possíveis conexões e calcular para qual das conexões comuta sem acrescentar um *overhead* significativo ao tempo total de transferência,

mesmo quando comparando com versões Java Socket comuns e mesmo com aplicações externas ao ambiente Java, neste caso o *WGet*.

Temos, no entanto, a noção que o correcto funcionamento de todo a *framework* desenvolvida tem as suas lacunas e limitações.

Começando pelo sistema onde os testes ocorreram, temos que abordar a capacidade que esta *framework* tem de ser portável. Esta capacidade está limitada, não devido à linguagem de programação utilizada (Java) que essa por si só tem um alto nível de portabilidade, mas devido ao *patch* instalado a nível do *kernel* do sistema. Este *patch*, apesar de fácil de instalar para quem conhece o processo, está limitado a um número reduzido de versões disponíveis, cada uma delas para um certo intervalo de versões de *kernel*. Ou seja, o mecanismo existente no *patch* pode mudar de uma versão para outra, o que poderia ter um impacto no correcto funcionamento do sistema. Só o facto de alguma das variáveis utilizadas mudar de nome ou contexto (valor), teria um impacto imprevisível nos mecanismos presentes na *framework*. Assim, este *framework* apenas se encontra com garantias de funcionamento para as versões utilizadas: *patch* (3.0); *kernel* (3.0.16); sistema operativo (*Ubuntu*, versão 11.10).

Existe também um número considerável de parâmetros que têm que ser afinados de modo a tirar total partido do padrão e *framework*. Como se verificou através dos testes (Teste de controlo, Subsecção 6.2.1), a aplicação desenvolvida com recurso ao padrão tem um papel crítico no funcionamento do sistema. Um programador que desenvolva uma aplicação que faça pedidos de um objecto por partes (blocos) tem que ter a consciência de que o tamanho de cada um desses pedidos tem um forte impacto no tempo total de transferência (Subsecção 6.2.1.1). Como foi possível verificar, um tamanho para esta parte mais pequeno implica um número maior de pedidos, um número maior de verificações de qualidade da conexão por parte do sistema, culminando com um maior tempo de transferência. Quem diz um ficheiro por partes pequenas, diz fazer a transferência de um número elevado de ficheiros de pequenas dimensões, e.g. *download* de todos os ficheiros numa certa directoria (que seria outra forma de aplicabilidade do padrão). Temos, no entanto, a plena consciência que este detalhe não depende do nosso padrão/*framework*, mas sim do uso que se faz do mesmo, logo serve o explicado anteriormente apenas como uma nota a ter em consideração para futuros desenvolvimentos.

Todos os parâmetros utilizados no *framework* que influenciem directamente o funcionamento do *NChannelSocket* merecem uma nota nesta conclusão: os valores de *timeouts* (Subsecção 6.2.4) para detectar falhas na conexão ou na leitura afectam o tempo de reacção do *NChannelSocket* na comutação de canais; os valores utilizados entre cada medição dos valores provenientes do *Web10g* (250ms, Secção 5.2) afecta, logicamente, o número de medições e qualidade dos dados medidos de cada conexão; os valores do parâmetro *alpha* utilizados para comparar os valores recentes com os valores passados (Subsecção 5.3.6) influenciam directamente a capacidade do sistema decidir correctamente se muda ou não de canal aquando de uma quebra na qualidade do mesmo; a probabilidade utilizada para a comutação aleatória (Subsecção 5.3.6) influência directamente a chance do

sistema comutar para outra conexão que seja considerada melhor, quando o canal estiver em bom funcionamento.

Independentemente de todas estas facetas que necessitam de estudo mais aprofundado é uma convicção que os objectivos deste trabalho foram cumpridos com sucesso, os testes validam o funcionamento do sistema e, apesar de alguns melhoramentos (Secção 7.2) e cuidados a ter em conta, estamos satisfeitos com a qualidade do trabalho desenvolvido.

7.2 Trabalho futuro

Quanto ao trabalho futuro, existem alguns aspectos que podem ser melhorados, principalmente no que toca aos valores dos diversos parâmetros referidos na secção anterior.

Um aspecto a salientar é a frequência na utilização das sondas (*probes*). Sempre que uma nova ligação é criada (no arranque ou na mudança), são criadas também sondas para todas as outras conexões que não estão a ser utilizadas (Subsecção 5.3.5). Este aspecto poderia provocar alguma latência em servidores mais fracos, ou se em larga escala (e.g. em algum sitio que exista dezenas de alternativas para o mesmo servidor) ser considerado uma espécie de ataque DoS (*denial of service* [dos]). Um melhoramento seria só abrir conexões como actualmente e fechá-las logo após uma medição, ou então só abrir novas conexões de X em X tempo.

Todos os parâmetros referidos necessitariam de uma afinação extra, com um processo de testes intensivo e cuidado, de modo a afinar mais correctamente os mecanismos da *framework* que dependem destes parâmetros.

O tamanho do pedido efectuado (tamanho do bloco) poderia ser recomendado através do próprio *framework* que, conforme o desempenho dos canais, sugeria a utilização de um tamanho do bloco que melhor se adequasse ao canal em uso. Teríamos que ter em atenção o tipo de aplicação desenvolvida, pois o significado deste parâmetro poderia mudar, e.g. poderia ter um significado diferente caso fosse um ficheiro por partes ou vários ficheiros completos.

Um possível melhoramento seria de definir os valores de *timeout* de forma dinâmica, conforme o desempenho actual dos canais. Se um canal tivesse com pior *performance* (e.g. menos *throughput*), poderíamos definir os valores de *timeout* para um número maior, em relação a um canal que tivesse com melhor *performance*, onde esse valor poderia baixar. Isto se quiséssemos dar mais tempo à conexão pior para tentar recuperar, pois o contrário também poderia ser aplicável. Dando menos tempo de *timeout* à medida que a conexão ficasse com pior desempenho, chegaríamos a um ponto em que seria lançada a excepção de *timeout* de leitura mais rapidamente, comutando para um canal mais estável.

Também o valor do parâmetro que influencia a mudança aleatória necessitaria de uma afinação extra. De igual modo, poderíamos melhorar esta probabilidade, tornando-a num valor dinâmico conforme a qualidade actual do canal em relação a outros canais disponíveis. No caso de termos certezas de que existe uma melhor conexão, talvez não

fosse má ideia atribuir a percentagem de 100% e forçando o sistema a mudar, em vez dos 60% actuais. Isto tornaria a mudança mais rápida, mas poderíamos cair no erro de mudar mais vezes para uma conexão que acabaria por não se relevar tão boa quanto o esperado. Tudo dependeria da capacidade do *NChannelSocket* avaliar correctamente os canais disponíveis.

O valor do parâmetro *alpha* utilizado como comparação entre as medições recentes e as do passado, de forma a determinar se a qualidade de um canal piorou, também necessitariam de um olhar mais atento. O valor actualmente definido (0.2) não foi sujeito a um controlo de qualidade extenso, pois isso implicaria um tempo alargado de estudo que não foi possível no tempo disponível. Um valor muito baixo para este parâmetro implicaria que à mínima alteração da qualidade mudaria de canal, o que poderia tornar-se contraproducente. Por outro lado, um número muito grande poderia implicar que esta alteração raramente iria acontecer.

Outra possível alteração seria a do próprio *framework* forçar a excepção em caso extremo, fechando o Socket em uso. Neste caso, comutaria de ligação de forma transparente para o utilizador, sem ser necessário uma chamada explicita ao método de teste de mudança (Subsecção 5.3.6). Isto poderia diminuir o tempo necessário à mudança de conexão, naqueles casos em que era necessário esperar pelo próximo pedido para testar uma quebra de qualidade. Contudo, poderia acabar por desencorajar alguns programadores pois deixariam de ter este factor de controlo.

Um dos últimos assuntos que queremos abordar nesta secção está relacionado com a extensibilidade do padrão desenvolvido, de modo a ser inserido num modelo em que impere não o *request-reply* simples (um canal activo), mas sim várias conexões em paralelo. Este modelo implica que o sistema teria que ser capaz de lidar com uma lista de conexões activas, em que teria que existir um modo de gestão dos parâmetros do sistema, e.g. uma aplicação que fizesse a transferência de um ficheiro em paralelo por partes, ou vários ficheiros completos em paralelo. De modo a suportar este tipo de aplicação um utilizador poderia requisitar ao *NChannelSocket* uma nova ligação (teria que existir uma espécie de *pool* de Sockets), num desenho tipo *master-worker* em que o *thread* do *NChannelSocket* seria responsável por lançar *threads* com os Sockets para cada pedido de nova transferência. Todo este mecanismo implicaria mudanças, algumas extensas, na *framework* e algumas alterações ao próprio padrão. Contudo, a ideia desenvolvida neste trabalho seria, com as devidas alterações, extensível a este novo modelo de transferência.

Por último, apresentamos um caso de possível desenvolvimento que merece alguma atenção. Este seria o de desenvolver uma aplicação com o *NChannelSocket* que utilize ligações seguras, e.g. SSL [FKK], HTTPS [Res], entre outros. Estes tipos de ligação segura implicam, na maioria dos casos, manter sessões abertas, partilhar chaves de encriptação e/ou transferir dados encriptados. Seria um caso de estudo interessante analisar o comportamento do *NChannelSocket* perante uma aplicação que utilizasse esses sistemas e ver como o mecanismo reagiria. Teríamos que ter em consideração que, ao mudar de conexão, a sessão poderia ser apagada e todo o mecanismo de autenticação teria que ser

reiniciado, o que aumentaria o *overhead* sobre o sistema. Este caso teria que ser merecedor de um estudo cuidado onde teriam que ser avaliadas as alterações necessárias a efectuar a todo o padrão e *framework* desenvolvido.

Bibliografia

[ABH10] Randall J. Atkinson, Saleem N. Bhatti, e Stephen Hailes. Evolving the internet architecture through naming. *IEEE Journal on Selected Areas in Communications*, pág. 1319–1325, 2010.

[ACK03] S. Agarwal, C.N. Chuah, e R.H. Katz. OPCA: robust interdomain policy routing and traffic control. In *Open Architectures and Network Programming, 2003 IEEE Conference on*, pág. 55–64, 2003.

[Al-06] F. Al-Shraideh. Host identity protocol. In *Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on*, pág. 203–203, 2006.

[AP] Mark Allman e Vern Paxson. Computing TCP\x27s retransmission timer. `http://tools.ietf.org/html/rfc2988`. Consultado em 2012-09-22.

[apa] The apache HTTP server project. `http://httpd.apache.org/`. Consultado em 2012-09-16.

[as611] AS65000 - BGP table statistics. `http://bgp.potaroo.net/as2.0/bgp-active.html`, Consultado em 2011-10-15, 2011-11-02 e 2011-12-11.

[BAB11] Begen, A., Akgul, T., e Baugher, M. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing*, pág. Volume: 15 , Issue: 2 Page(s): 54 – 63, Abril 2011.

[bgp11] The BGP instability report. `http://bgpupdates.potaroo.net/instability/bgpupd.html`, Consultado em 2011-12-11.

[Bon11] O. Bonaventure. TCP extensions for multipath operation with multiple addresses draft-ietf-mptcp-multiaddressed-03. 2011.

[brk04] LISP - a next generation routing architecture. `http://lisp.cisco.com/BRKRST-3045_Vegas2011.pdf`, Consultado em 2011-12-04.

[bui] Building a Web10G kernel . http://www.web10g.org/index.php?option=com_content&view=article&id=22&Itemid=57. Consultado em 2012-05-27.

[Cis] Cisco. BGP case studies - document ID: 26634. http://www.cisco.com/image/gif/paws/26634/bgp-toc.pdf. Consultado em 2012-01-23.

[CL05] R.K.C. Chang e M. Lo. Inbound traffic engineering for multihomed ASs using as path prepending. *Network, IEEE*, 19(2):18–25, 2005.

[CNJ+] A. Cabellos, A.R. Natal, L. Jakab, V. Ermagan, P. Natarajan, e F. Maino. LISPmob: mobile networking through LISP.

[Coh] Bram Cohen. The BitTorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html. Consultado em 2012-09-20.

[CR10] M. Carbone e L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[CRR98] C. Villamizar, R. Chandra, e R. Govindan. BGP route flap damping. http://www.ietf.org/rfc/rfc2439.txt, Novembro 1998. Consultado em 2012-09-23.

[DD08] A. Dhamdhere e C. Dovrolis. Ten years in the evolution of the internet ecosystem. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pág. 183–196, 2008.

[dos] CERT/CC denial of service. http://www.cert.org/tech_tips/denial_of_service.html. Consultado em 2012-09-21.

[FFM+11] D. Farinacci, V. Fuller, D. Meyer, D. Lewis, e cisco Systems. Locator/id separation protocol (lisp). internet draft (work in progress) draft-ietf-lisp-16. *IEEE - Internet Engineering Task Force - Network Working Group*, Novembro de 2011.

[FKK] Alan Freier, Paul Kocher, e Philip Karlton. The SSL protocol version 3.0. http://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00. Consultado em 2012-09-23.

[FLMW11] D. Farinacci, D. Lewis, D. Meyer, e C. White. LISP mobile node. *Network Working Group - Internet-Draft*, Outubro de 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GPSH12] S. Gebert, R. Pries, D. Schlosser, e K. Heck. Internet access traffic measurement and analysis. *Traffic Monitoring and Analysis*, pág. 29–42, 2012.

[GTK+99] Gordon Dodrill, Tim Ward, Karen Ward, Mark Harvey, Kathy Morton, e Eric Lindsay. C language tutorial. http://phy.ntnu.edu.tw/{~}cchen/ctutor.pdf, Março 1999.

[HA06] G. Huston e G. Armitage. Projecting future IPv4 router requirements from trends in dynamic BGP behaviour. In *Proc. of ATNAC*, 2006.

[HAS02] T. J Hacker, B. D Athey, e J. Sommerfield. Experiences using web100 for end-to-end network performance tuning. In *Proceedings of the 4th Visible Human Project Conference*, 2002.

[htm] HTML - hypertext markup language. http://www.w3.org/TR/REC-html40/. Consultado em 2012-09-23.

[htt] HTTP/1.1: header field definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html. Consultado em 2012-09-23.

[Hus02a] Geoff Huston. BGP growth revisited. http://www.potaroo.net/ispcol/2011-11/bgp2011.pdf, Consultado em 2011-11-02.

[Hus02b] Geoff Huston. The BGP world is flat! http://www.potaroo.net/ispcol/2011-12/flat.pdf, Consultado em 2011-11-02.

[Hus06] G. Huston. *Internationalizing the Internet*. Retrieved, 2006.

[iet] Internet engineering task force (IETF). http://www.ietf.org/. Consultado em 2012-09-23.

[java] Enum types (The java™ tutorials > learning the java language > classes and objects). http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html. Consultado em 2012-09-23.

[javb] How garbage collection works in java. http://javarevisited.blogspot.pt/2011/04/garbage-collection-in-java.html. Consultado em 2012-09-23.

[javc] InetAddress (Java platform SE 7), java api. http://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html. Consultado em 2012-09-23.

[javd] InetSocketAddress (Java platform SE 7), java api. http://docs.oracle.com/javase/7/docs/api/java/net/InetSocketAddress.html. Consultado em 2012-09-23.

[jave] java.com: Java + you. http://www.java.com/pt_BR/. Consultado em 2012-09-23.

[javf] java.net (Java platform SE 7), java api. http://docs.oracle.com/javase/7/docs/api/java/net/package-frame.html. Consultado em 2012-09-23.

[javg] Lesson: Exceptions (The java™ tutorials > essential classes). http://docs.oracle.com/javase/tutorial/essential/exceptions/. Consultado em 2012-09-23.

[javh] Socket (Java platform SE 7), java api. http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html. Consultado em 2012-09-23.

[javi] System (Java platform SE 7). http://docs.oracle.com/javase/7/docs/api/java/lang/System.html. Consultado em 2012-09-16.

[javj] The try-with-resources statement (The java™ tutorials > essential classes > exceptions). http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. Consultado em 2012-09-23.

[JMY+08] D. Jen, M. Meisel, H. Yan, D. Massey, L. Wang, B. Zhang, e L. Zhang. Towards a new internet routing architecture: arguments for separating edges from transit core. In *7th ACM Workshop on Hot Topics in Networks (HotNets), Calgary, Alberta, Canada*, 2008.

[jnaa] JNA - GitHub. https://github.com/twall/jna. Consultado em 2012-09-18.

[jnab] JNA frequently asked questions. https://github.com/twall/jna/blob/master/www/FrequentlyAskedQuestions.md. Consultado em 2012-09-18.

[jnia] Developing a java native interface (JNI). http://www.javahotchocolate.com/tutorials/jni.html. Consultado em 2012-09-18.

[jnib] Java SE 7 java native interface-related APIs and developer guides. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/. Consultado em 2012-09-18.

[Kal] Shiv Kalyanaraman. Load balancing in BGP environments using online simulation and dynamic NAT. http://www.caida.org/workshops/isma/0112/talks/shiv/.

[Kar] Daniel Karrenberg. Dns root name servers explained for non-experts. http://www.isoc.org/briefings/019/briefing19.pdf.

[Ken03] S.T. Kent. Securing the border gateway protocol. *The Internet Protocol Journal*, 6(3):2–14, 2003.

[LABJ00] C. Labovitz, A. Ahuja, A. Bose, e F. Jahanian. Delayed internet routing convergence. *ACM SIGCOMM Computer Communication Review*, 30(4):175–187, 2000.

[LFFM11] D. Lewis, V. Fuller, D. Farinacci, e D. Meyer. Interworking LISP with IPv4 and IPv6. 2011.

[lis04a] LISP part 1: Problem statement, architecture and protocol description - YouTube. http://www.youtube.com/watch?v=WSl1RAlFU3s, Consultado em 2011-12-04.

[lis04b] LISP part 2 - mapping database infrastructure and interworking - YouTube. http://www.youtube.com/watch?v=_bz4cRuAcak, Consultado em 2011-12-04.

[lis04c] LISP part 3 - deployed network and Use-Cases - YouTube. http://www.youtube.com/watch?v=fxdm-Xouu-k, Consultado em 2011-12-04.

[mak] make - linux command - unix command. http://linux.about.com/library/cmd/blcmdl1_make.htm. Consultado em 2012-09-18.

[Mal] Gary Scott Malkin. RIP version 2. http://tools.ietf.org/html/rfc2453. Consultado em 2012-09-23.

[Mee] ICANN Meeting. Anycasting and the rootservers. http://archive.icann.org/en/meetings/vancouver/jlc-anycasting.pdf.

[MFFL] David Meyer, Vince Fuller, Dino Farinacci, e Darrel Lewis. LISP alternative topology (LISP+ALT). http://tools.ietf.org/html/draft-fuller-lisp-alt-05. Consultado em 2012-09-23.

[MSRH10] Matt Mathis, Jeffrey Semke, Raghu Reddy, e John Heffner. Documentation of variables for the web100 TCP kernel instrumentation set (KIS) project. http://www.web100.org/download/kernel/tcp-kis.txt, 2010.

[nat] HowStuffWorks "How network address translation works". http://computer.howstuffworks.com/nat.htm. Consultado em 2012-09-20.

[NSS10] E. Nygren, R. K. Sitaraman, e J. Sun. The akamai network: A platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[Phi06] Philip Smith. BGP best practices. http://www.ripe.net/ripe/meetings/regional-meetings/manama-{2006/BGPBCP%.pdf}, Novembro 2006. Consultado em 2012-09-23.

[PMFR11] Sun Peng, Yu Minlan, Micheal J. Freedman, e Jennifer Rexford. Identifying performance bottlenecks in CDNs through TCP-Level monitoring. 2011.

[Posa] J. Postel. Internet protocol. http://tools.ietf.org/html/rfc791. Consultado em 2012-09-22.

[Posb] J. Postel. The TCP maximum segment size and related topics. https://tools.ietf.org/html/rfc879. Consultado em 2012-09-22.

[QIDLB07] B. Quoitin, L. Iannone, C. De Launois, e O. Bonaventure. Evaluating the benefits of the locator/identifier separation. In *Proceedings of 2nd ACM/IEEE international workshop on Mobility in the evolving internet architecture*, pág. 5, 2007.

[Res] E. Rescorla. HTTP over TLS. http://tools.ietf.org/html/draft-ietf-tls-https-03. Consultado em 2012-09-23.

[RHF09] Costin Raiciu, Mark Handley, e Alan Ford. Multipath TCP design decisions. http://www.cs.ucl.ac.uk/staff/C.Raiciu/files/mtcp-design.pdf, 2009.

[Riz] Luigi Rizzo. Dummynet. http://info.iet.unipi.it/{~}luigi/dummynet/. Consultado em 2012-09-16.

[RUJ+99] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, e T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt, Junho 1999. Consultado em 2012-09-21.

[SAP] W. Richard Stevens, Mark Allman, e Vern Paxson. TCP congestion control. http://tools.ietf.org/html/rfc2581. Consultado em 2012-09-22.

[Sau11] Damien Saucez. Mechanisms for interdomain traffic engineering with lisp. *ICTEAM, Louvain School of Engineering, Université catholique de Louvain, Louvain-la-Neuve Belgium*, 2011.

[SC91] T. Socolofsky e C. Kale. A TCP/IP tutorial. http://www.ietf.org/rfc/rfc1180.txt, Janeiro 1991. Consultado em 2012-09-23.

[SF11] M. Scharf e A. Ford. MPTCP application interface considerations. 2011.

[Ste] W. Richard Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. http://tools.ietf.org/html/rfc2001. Consultado em 2012-09-22.

[Ste07] R. Stewart. Stream control transmission protocol. 2007.

[sys] sys_attrs_inet(5). http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V50A_HTML/MAN/MAN5/0164____.HTM.

[tcp] TCP specific programming information. http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V50_HTML/ARH9UATE/TRNSSPCF.HTM.

[tcp12] TCP selective acknowledgment options (SACK). http://opalsoft.net/qos/TCP-90.htm, Consultado em 2011-12-12.

[TDG] Technology Development Group Technology Development Group. TCP/IP performance over satellite links - summary report. http://www.adec.edu/nsf/tcpip-performance.pdf.

[ubua] Home | ubuntu. http://www.ubuntu.com/. Consultado em 2012-09-20.

[ubub] Ubuntu linux - slow wireless network speed - force adapter speed - PenguinTutor linux blog. http://www.penguintutor.com/blog/viewblog.php?blog=4414. Consultado em 2012-09-19.

[ubuc] Ubuntu manpage: ipfw – IP packet filter and traffic accounting. http://manpages.ubuntu.com/manpages/precise/en/man4/ipfw.4freebsd.html. Consultado em 2012-09-15.

[ubud] Ubuntu manpage: tcp - TCP protocol. http://manpages.ubuntu.com/manpages/oneiric/man7/tcp.7.html. Consultado em 2012-09-17.

[ubue] Ubuntu manpage: Wget - the non-interactive network downloader. http://manpages.ubuntu.com/manpages/oneiric/man1/wget.1.html. Consultado em 2012-09-13.

[vB09] I. van Beijnum. One-ended multipath TCP. 2009.

[weba] Web10G. http://web10g.org/. Consultado em 2012-09-20.

[webb] Web10G kernel patch and API packages. http://web10g.org/index.php?option=com_remository&Itemid=65&func={se%lect}&id=4. Consultado em 2012-09-23.

[webc] Web10G kernel patch for linux 3.0. http://web10g.org/index.php?option=com_remository&Itemid=65&func=fileinfo&id=20. Consultado em 2012-09-20.

[YCB07] X. Yang, D. Clark, e A. W Berger. NIRA: a new Inter-Domain routing architecture. *IEEE/ACM Transactions on Networking*, 15(4):775–788, Agosto 2007.

[ZGC03] Dapeng Zhu, Mark Gritter, e David R. Cheriton. Feedback based routing. *SIGCOMM Comput. Commun. Rev.*, 33(1):71–76, Janeiro 2003.